UNIVERSITAT POLITÈCNICA DE CATALUNYA

Study, design and implementation of robust entropy coders

by

Marcial Clotet Altarriba

in the

Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona

Departament de Física Aplicada

Advisor: Enrique Garcia-Berro Montilla

Co-advisor: Alberto González Villafranca

July 2010

Acknowledgements

En primer lloc, m'agradaria agraïr al meu tutor Enrique Garcia-Berro l'ajuda durant la realització d'aquest projecte. Igualment imprescindibles han estat els consells d'en Jordi Portell i l'Alberto G. Villafranca. Puc dir sincerament que ha estat un plaer i un honor poder treballar amb vosaltres durant aquest temps.

Gràcies als meus pares i al meu germà. Ells han estat sempre encoratjadors i a la vegada comprensibles. Part del mèrit d'aquest projecte, que tanca un cicle, us correspon a vosaltres. Sense el vostre suport, confiança i paciència mai hauria arribat fins aquí.

Una menció especial es mereix la Nuria. Gràcies per estar sempre al meu costat, per la teva paciència i pel teu ajut. Però sobretot per deixar-me compartir la vida amb tu i fer-me cada dia més feliç. Gràcies amor.

Finalment, agraïr als meus amics el seu suport. Als de la UPC, als d'Igualada, al Bernat i la Cristina, al David i la Miriam, i tots els que sempre em feu costat. Vosaltres, tots, heu donat sentit a aquesta carrera i m'heu ajudat a continuar. Gràcies.

Contents

List of Figures

List of Tables

Abbreviations

AF Astrometric Field

BP Blue Photometers

CCSDS Consultative Committee for Space Data Systems

CLDCR CCSDS 121.0 Lossless Data Compression Recommendation

DS Doubled-Smoothed

FAPEC Fully Adaptative PEC

FELICS Fast, Efficient, Lossless Image Compression System

FITS Flexible Image Transport System

FOCAS Fiber-Optics Communications for Aerospace Systems

FS Fundamental Sequence

GIBIS Gaia Instrument and Basic Image Simulator

GPS Global Positioning System

HyPER Hybrid PEC/REGLIUS

ITU International Telecommunication Union

LC Large Coding

LE Low Entropy

LSB Least Significant Bits

MSB Most Significant Bits

REGLIUS Rice-Exponential Golomb, LImited, with reUsed Stopbit

PDF Probability Density Function

PEC Prediction Error Coder

PEM Prediction Error Mapper

PGM Portable Gray Map

RP Red Photometers

RVS Radial Velocity Spectrometer

SE Second Extension

SM Sky Mapper

SNR Signal to Noise Ratio

ZB Zero Block

Chapter 1

Introduction

Data compression systems for satellite payloads have several tight restrictions. First, the data block size should be kept rather small in order to avoid losing large amounts of data if transmission errors occur [1]. More precisely, data should be compressed in small independent data blocks. This is at odds with the fact that most adaptive data compression systems perform optimally only after a large amount of data is processed. Secondly, the processing power for software implementations (or electrical power, in hardware implementations) is limited in space. Therefore, the compression algorithm should be as simple and quick as possible. Finally, the required compression ratios are increasing as new missions which handle huge amounts of data are conceived and launched. When all these restrictions are combined with the need of a lossless operation, the design of such a data compression system becomes a true challenge.

The CCSDS issued its recommendation for lossless data compression [2] in 1993 with the intention of offering a solution to data compression requirements in space missions. The proposed solution is a very simple (thus quick) algorithm that operates in blocks of just 8 or 16 samples. This recommendation has been used in several missions [3] including hardware implementations [4]. In fact the CCSDS 121.0 recommendation has been the "de facto" standard in these scenarios. This is due to the reasonable compression ratios achieved with low processing requirements.

Despite its powerful features, this standard compression system is not exempt of problems either. The critical problem arises at the coding stage, as the Rice algorithm is not intended to compress noisy data. In fact, its efficiency abruptly decreases when noise

is introduced in the data. This is a major issue since most space-based measurements are contaminated with noise and outliers. Therefore, the CCSDS 121.0 recommendation is not an optimum solution in most of the cases.

In this work we explore the concept of outlier-resilient entropy coders, looking for a better solution than that of the CCSDS 121.0 standard. The goal is to offer a data compression solution suitable for space systems with the best possible compression results, even in case of data contaminated with noise and outliers. First, a simple change in the CCSDS 121.0 coding stage is proposed. More specifically, we study the substitution of the Rice coder by an exponential or subexponential coder, keeping the rest of the recommendation unchanged. However, the CCSDS standard adaptive framework has other inherent limitations. Therefore, a completely different approach will be sought as well. Inspired on by previous solutions such as the PEC coder, a segment coding strategy will be used for the compressor while including a newly devised coding strategy which incorporates desirable features of both Rice and exponential codes.

This report is organized as follows. In chapter 2 the limitations of the Rice coder and its effects on CCSDS 121.0 standard are studied. Chapter 3 describes the exponential-Golomb coder and its implementation within the CCSDS 121.0 framework, and it discusses the results obtained on synthetic data. Chapter 4 follows the same approach but for the subexponential coder. A different and new approach to the data compression problem is proposed in chapter 5 with the description of the HyPER coder based on REGLIUS codes. The final comparison between all the implemented coders and the current standard using both synthetic and real data is shown in chapter 6. Finally, in chapter 7 we summarize our major findings, we draw our conclusions and we propose several lines of future work.

Chapter 2

Context

2.1 CCSDS 121.0 Lossless Data Compression Recommendation

This chapter presents the compressor structure defined in the CCSDS 121.0 Lossless Data Compression Recommendation (CLDCR) standard. First, the general architecture is described paying special attention to the pre-processing and the adaptive stage. In second place the Rice coder is introduced in detail.

2.1.1 CCSDS architecture

The CCSDS standard recommends a two-stage architecture, namely, a pre-processing stage followed by an entropy coder. This is an otherwise typical solution used in several systems, as discussed in [5] or [1]. The recommendation does not strictly specify the pre-processing stage, since it must be carefully tailored for each mission in order to achieve the best ratios. Figure 2.1 shows the two functions contained in the pre-processor, namely, prediction and mapping.

The pre-processor subtracts the predicted value from the current value. The resulting prediction error is then mapped to a positive integer value. When a predictor is adequately chosen, the prediction errors tend to be small and thus they can be coded with fewer bits. Typically, the prediction errors follow a probability distribution approaching a Laplacian. This is the optimal case as the recommendation is designed to work with

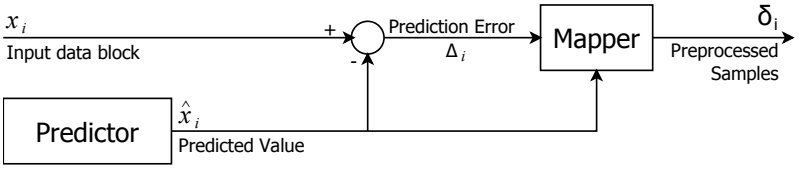

FIGURE 2.1: CCSDS preprocessing structure.

such distribution. The unit-delay predictor is the most basic approach for this stage, although more complex solutions exist — or can be designed for each case if necessary.

The second stage is based on the Rice coder [6] with an adaptive layer that selects the most suitable k parameter for each data block. For very low entropy levels, other coding procedures such as the Zero Block (ZB), Second Extension (SE) or Fundamental Sequence (FS) options [7] are selected automatically, boosting the compression level beyond the capabilities of the Rice compressor. Figure 2.2 shows the adaptive entropy coder structure with a pre-processor. The adaptive stage chooses the best among a set of code options to represent an incoming block of pre-processed data samples. Specifically, it determines the total length of the coded block considering the available options (including Rice coding with $k = 1$ to $k = 13$) and then it selects the option leading to the shortest total length. A unique identifier for each option is added to every coded sequence. This indicates to the decoder which decoding option must be used.

2.1.2 Rice coder

Rice codes are optimal for data following discrete Laplacian (or two-sided geometric) distributions [8], which are expected to occur after the CCSDS 121.0 pre-processing stage [2] — or, in general, after any adequate pre-processing stage. However, this assumes a correct operation of the predictor, which cannot be taken for granted as noisy samples and outliers can modify expected distribution.

It is known that Rice codes are a special case of more general Golomb codes where the parameter m is a power of 2, $m = 2^k$, with $k \geq 0$. Rice codes have 2^k codes of length, starting with a minimum length of $k + 1$. A significant feature of Rice codes is that it is a very simple coding algorithm. Once the parameter k has been defined, the code is easily constructed by simply separating the k Least Significant Bits (LSB) of the

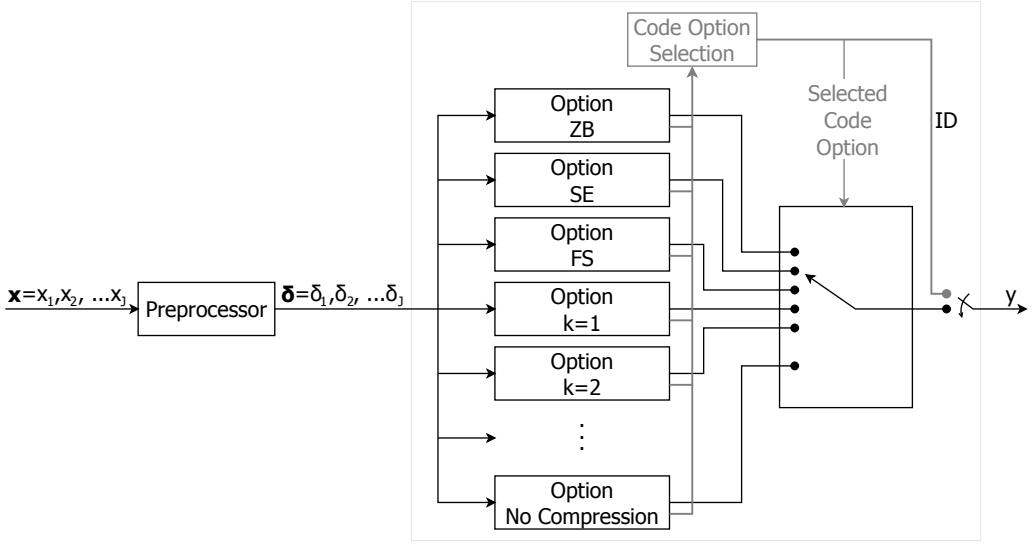

FIGURE 2.2: CCSDS adaptive stage.

integer n which will become the LSB of the code. These will follow the $j = \lfloor \frac{n}{2^k} \rfloor$ bits in unary code. These codes are easily constructed with few operations which are not computationally expensive. This is an important feature, as computing power is scarce in space applications. Finally, under the CCSDS framework it is required to compute the length of a given code constantly, thus a simple equation is desired. Suitably, the length of a Rice code for an integer n coded using a parameter k can be easily computed as $1 + k + \lfloor \frac{n}{2^k} \rfloor$.

The k parameter of a Rice coder must be chosen carefully in order to obtain the expected compression ratios for a given set of data. Table 2.1 illustrates some Rice codes for small values and low k configurations. Note the rapid increase in code length for small values of k — although such low k values provide the shortest codes for small values. If Rice codes were used statically (that is, manually calibrating the k parameter using simulations), an unacceptable risk would appear. It might occur that the expected data set only has low values, and thus a low k is chosen, for instance $k = 1$. With this configuration, receiving a single high value (or outlier) such as 20000 would lead to an output code of about ten thousand bits. This flawed behavior is the reason why the CCSDS standard introduced the adaptive layer to automatically select the best k for each data block. Note that $k = 0$ is not considered in the recommendation, since it coincides with the FS option already included in CCSDS 121.0. This automatic calibration significantly reduces the effect of outliers present in the data gathered in space missions, leading to acceptable

n	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
0	0|	0|0	0|00	0|000	0|0000	0|00000
1	10|	0|1	0|01	0|001	0|0001	0|00001
2	110|	10|0	0|10	0|010	0|0010	0|00010
3	1110|	10|1	0|11	0|011	0|0011	0|00011
4	11110|	110|0	10|00	0|100	0|0100	0|00100
5	111110|	110|1	10|01	0|101	0|0101	0|00101
6	1111110|	1110|0	10|10	0|110	0|0110	0|00110
7	11111110|	1110|1	10|11	0|111	0|0111	0|00111
8	111111110|	11110|0	110|00	10|000	0|1000	0|01000
9	1111111110|	11110|1	110|01	10|001	0|1001	0|01001
10	11111111110|	111110|0	110|10	10|010	0|1010	0|01010
11	111111111110|	111110|1	110|11	10|011	0|1011	0|01011
12	1111111111110|	1111110|0	1110|00	10|100	0|1100	0|01100
13	11111111111110|	1111110|1	1110|01	10|101	0|1101	0|01101
14	111111111111110|	11111110|0	1110|10	10|110	0|1110	0|01110
15	1111111111111110|	11111110|1	1110|11	10|111	0|1111	0|01111
16	11111111111111110|	111111110|0	11110|00	110|000	10|0000	0|10000

TABLE 2.1: Rice-Golomb codes for values 0 to 16 and $k=0$ to $k=5$.

ratios even with rapidly changing statistics. Nevertheless, this is done by increasing the value of the parameter when such outliers are found. For instance, in a data block where all the values are small (or even zero), a single high value makes CCSDS 121.0 select a high value of k, thus leading to a small compression ratio. The goal of this project is to reduce the effect of such outliers even within a data block, making possible to select smaller k values and, thus, increasing the compression ratios.

2.2 Other existing solutions

The Rice codes are adequate when the compressed data follows a geometric statistical distribution, which often arises after an adequate pre-processing stage. However, any deviation from this statistic can lead to a significant decrease of the final compression ratio. The PEC solution was devised in previous studies [9]. It is focused on the compression of signed prediction errors, and hence a pre-processing stage based on a data predictor plus a differentiator (outputting signed values) is mandatory. Nevertheless, other pre-processing stages outputting signed values close to zero may be used as well.

PEC is composed of three coding options, namely, Low Entropy (LE), Doubled-Smoothed (DS) and Large Coding (LC). All these are segmented variable-length codes. Figure

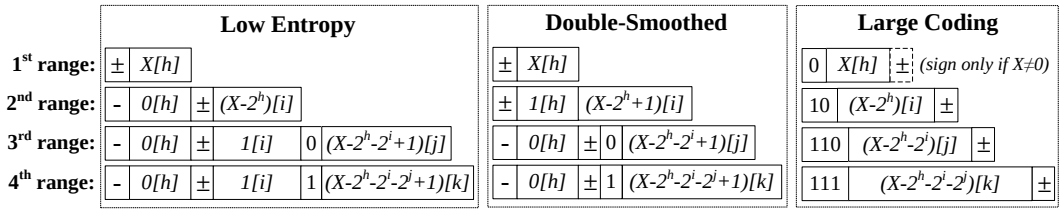

FIGURE 2.3: The three PEC coding strategies.

2.3 offers a schematic view of the coding strategy used in PEC. The coding scheme is completely different from the Rice coder. The three coding options share the same principles: the range of the data to be coded is split into four smaller ranges (or segments). The size of each segment determines its corresponding coding parameter (h, i, j or k), which indicates the number of bits required to code the values of that segment. This set of parameters is called *coding table* and they are independent each other.

For each coded value the appropriate segment is chosen and the adequate number of bits is used. PEC assumes that values are close to zero. However, one of the main advantages of this coding strategy is that it is flexible enough to adapt to data distributions with probability peaks far from zero. With an adequate choice of parameters, good compression ratios can still be reached with such distributions. PEC can be considered a partially adaptive algorithm. That is, the adequate segment (and hence the code size) is selected for each one of the values. This is obviously an advantage with respect to the Rice coder, which uses a fixed parameter for all the values — at least within a given coding block, in the case of the CCSDS recommendation. Another advantage with respect to Rice is that PEC limits the maximum code length to twice the symbol size in the worst case. Nevertheless, despite these features, PEC must be calibrated for each case in order to get the best compression ratios.

An adaptive version of PEC which solves the previously commented weaknesses is also available [10]. This is called Fully Adaptative PEC (FAPEC). It adds an adaptive layer to PEC in order to configure its coding table and coding option according to the statistics of each data block. Nearly optimal compression results can be achieved without the need of any preliminary configuration and without previous knowledge of the statistics of the data to be compressed. FAPEC was designed with the quickest possible operation in mind, even at the expense of a slight decrease in the compression ratio. The adaptive stage accumulates the values to be coded while building a histogram of their modules. This is a logarithmic-like histogram, where higher values are grouped and

mapped to fewer bins. This procedure reduces the memory required for the histogram. An algorithm analyzes the histogram and determines the best coding option and coding table. Once the coding option and the corresponding parameters have been determined, they are output as a small header followed by all the FAPEC codes for the values of that block. By explicitly indicating the FAPEC configuration we make possible to change the FAPEC decision algorithms without requiring any modification in the receiver.

The approach followed by PEC and FAPEC has proved to offer excellent results, adapting very well to noise and outliers in the data — even with large fractions of these. Therefore, an improved segmented coding strategy may be interesting to investigate. This will be the purpose of chapter 5.

Chapter 3

Exponential Golomb coder

3.1 Interest in exponential coders

The CLDCR has some important limitations, namely, a decrease in its compression efficiency when dealing with noisy data, non-Laplacian distributions, or data contaminated with outliers, in general [10]. This is caused by the high sensitivity of Rice codes to such outliers. On the other hand, there are other Golomb codes. The length of these codes grows slowly in case of outliers. Exponential Golomb codes are an example [11]. For this reason, we find interesting to implement and test an exponential Golomb coder within the CLDCR compressor structure in order to enhance its resiliency against noise and outliers with minimal changes in the current architecture. Keeping such changes as minimal as possible benefits the outcome, since the CLDCR compressor structure has been reliably tested in multiple missions [3]. A totally different and new compressor structure such as FAPEC [10] may require more time and resources for being tested and assessed for space applications.

Rice codes are optimal for data with discrete Laplacian (or two-sided geometric) distributions [8], which are expected after the CLDCR pre-processing stage [2] — or, in general, after any adequate pre-processing stage. However, this assumes a correct operation of the predictor which cannot be taken for granted as noisy samples and outliers can modify the expected distribution. This is specially true for the space environment, where prompt particle events (such as cosmic rays or solar protons) will affect the on-board instrumentation. Any deviation from the expected statistic can lead to a significant

decrease in the resulting compression ratio. This is the case of the the data passed to the compressor in the CCSDS 121.0 standard. Ideally, the values that reach the coder are close to zero as the samples are pre-processed by a predictor before the coder. The definition of the predictor is not part of the standard, and it must be tailored for each mission as it depends on the nature of the data sources. If correctly defined, when the predictor works properly the prediction error tends to be small and has a probability distribution function that approaches a Laplace distribution [1, 2, 8]. However, if the predictor does not work properly (due to, for instance, outliers resulting from cosmic rays), the CCSDS compressor performance drops abruptly.

The main reason for the CCSDS performance to drop abruptly when noise is introduced is that Rice codes are not intended to be used with noisy data. This limitation is due to the fact that the length of Rice codes grows too fast for large values, specially when low values are assigned to the k parameter. Appropriately, exponential Golomb codes provide shorter lengths than Rice codes for large values. However, smooth code growth for small data values provided by the Rice codes is lost. Whether the compression gain in larger values is more relevant than the loss in lower values will determines whether the exponential Golomb coder is suitable or not for this application.

3.2 Theoretical basis of exponential Golomb codes

The main feature of the exponential Golomb codes is that the number of codewords with length L grows exponentially with L. This property allows these codes to perform well for exponential probability distributions with larger dispersions.

As in the case of Rice codes, the exponential Golomb codes depend on a nonnegative parameter m. In this case, m is determined as $m = 2^k$. Therefore, only the parameter k must be specified to obtain m. This parameter k will also indicate the length of the suffix for the code. Exponential Golomb codes have three different parts which, once concatenated, produce the code. Two intermediate values are used to build the code, f and w, which are shown in Eqs. (3.1) and (3.2). The first part is the unary code of f. After this, the f LSB of w coded in plain binary are concatenated. Finally, the k LSB of the original value n are added. Detailed steps of how to implement the exponential Golomb coder are provided in section 3.3.

$$\begin{cases} k = 2_d \\ n = 42_d = 101010_b \end{cases} \Rightarrow \begin{cases} w = 1 + \left\lfloor \frac{n}{2^k} \right\rfloor = 1 + \left\lfloor \frac{42}{2^2} \right\rfloor = 11_d = 1011_b \\ f(42_d) = \left\lfloor \log_2(1 + \frac{n}{2^k}) \right\rfloor = \left\lfloor \log_2(1 + \frac{42}{2^k}) \right\rfloor = 3_d = 11_b \end{cases}$$

$$\underbrace{\underbrace{\text{Unary code of } f(n)}_{1110_b} + \underbrace{f(n) \text{ LSB of w}}_{011_b} + \underbrace{k \text{ LSB of n}}_{10_b}}_{111001110_b = 462_d}$$

FIGURE 3.1: Exponential Golomb coding example.

$$w(n) = 1 + \left\lfloor \frac{n}{2^k} \right\rfloor \tag{3.1}$$

$$f(n) = \left\lfloor \log_2(1 + \frac{n}{2^k}) \right\rfloor \tag{3.2}$$

In figure 3.1 a coding example for the exponential Golomb algorithm is shown. In this example, the value n is 42 (101010 in binary), and $k = 2$. Unary coding is shown as n ones followed by a zero stop-bit, although ones and zeroes are interchangeable without loss of generality. The result has been coded with 9 bits, while if the Rice coder is used instead it would have resulted in a 13-bits code. Assuming that the original value was coded using 16 bits, a noticeable compression of the original data has thus been successfully achieved. In table 3.1 some exponential Golomb codes are presented for n up to 16 and k up to 5.

The difference between the length of a Rice code and an exponential Golomb code grows with n. As an example, consider a large 16-bit value, $n = 65535$. Even using the highest k parameter, $k = 13$, the Rice coder would produce a codeword with 21 bits while the exponential Golomb code would lead to 20 bits. With lower k values, this difference becomes much larger: with $k = 10$, Rice would output 67 bits in that case, while the exponential Golomb coder would output just 21 bits. The length difference between both coders for different values of k is shown in figure 3.2.

The CLDCR compressor structure has a no-compression option which is used in the most extreme cases to avoid expanding data. By design, this no-compression strategy is activated when the length of the coded block with any available strategy exceeds the original block length when coded with standard binary. Thus, even with such a bad

n	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
0	0	00	000	0000	00000	000000
1	100	01	001	0001	00001	000001
2	101	1000	010	0010	00010	000010
3	11000	1001	011	0011	00011	000011
4	11001	1010	10000	0100	00100	000100
5	11010	1011	10001	0101	00101	000101
6	11011	110000	10010	0110	00110	000110
7	1110000	110001	10011	0111	00111	000111
8	1110001	110010	10100	100000	01000	001000
9	1110010	110011	10101	100001	01001	001001
10	1110011	110100	10110	100010	01010	001010
11	1110100	110101	10111	100011	01011	001011
12	1110101	110110	1100000	100100	01100	001100
13	1110110	110111	1100001	100101	01101	001101
14	1110111	11100000	1100010	100110	01110	001110
15	111100000	11100001	1100011	100111	01111	001111
16	111100001	11100010	1100100	101000	1000000	010000

TABLE 3.1: Some exponential Golomb codes.

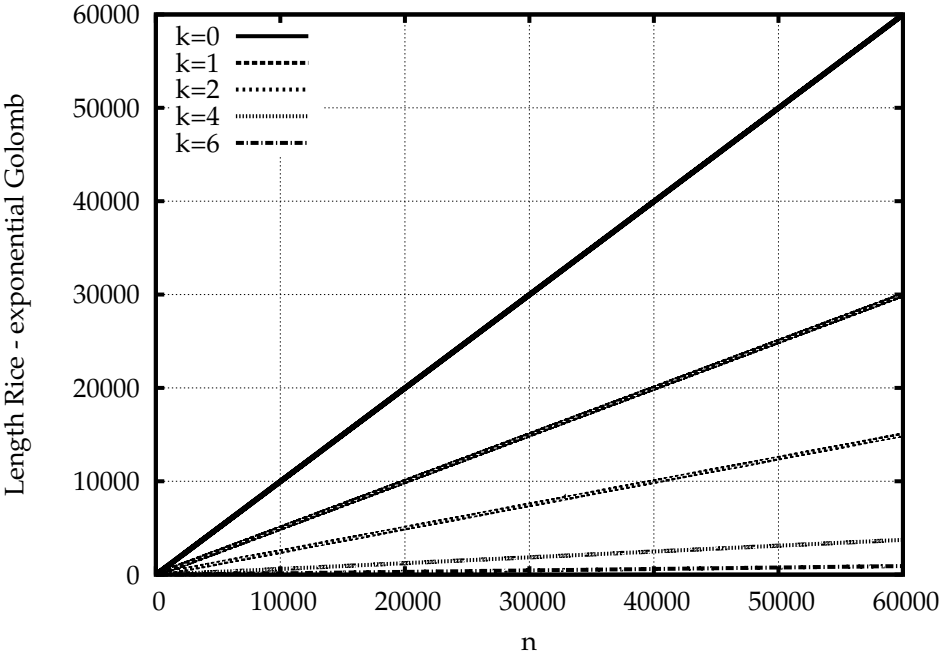

FIGURE 3.2: Code length difference between Rice exponential-Golomb.

performance of Rice codes on large values, the CLDCR will never expand the data — at least not significantly. However, it is clear that even a single large value within a data block will degrade the overall performance of the original coder.

Within the CLDCR compressor structure, it is required to compute the length of the code for each given n and k. This information is used to adequately choose the best coding strategy, or the best value of k, based on the length of the produced block length [2]. For the exponential Golomb coder, the length of the code can be obtained using Eq. (3.3). Since the logarithm is truncated, the code length increases by 2 bits each time the logarithm increases by 1, so $1 + \frac{n}{2^k}$ is a power of 2.

$$l(n) = 1 + 2f(n) + k = 1 + 2\lfloor \log_2[1 + \frac{n}{2^k}] \rfloor \tag{3.3}$$

As a side note, it is worth mentioning that exponential Golomb codes can be generalized by replacing $m = 2^k$ by an arbitrary positive integer. These codes are called generalist exponential Golomb codes. However, from an implementation perspective, $m = 2^k$ is preferred as it has a lower computational cost. It also worth mentioning that the exponential Golomb codes are equivalent to the triplet (1, s, ∞) of start-stop codes [12]. Finally, it must be noted that bidirectional versions for both the Rice and exponential Golomb codes exist [13]. These codes have the same length as the original, one-directional codes with the additional property that they can be decoded in both directions. These codes have been adopted by the International Telecommunication Union (ITU) for use in the video coding parts of `MPEG-4`, specially in the H.263v2 and H.264 standards [14].

3.3 Practical implementation

While multiple algorithms of exponential Golomb can be found, in this section an implementation of the coder as described in [12] is discussed. The steps to code a nonnegative value n with the parameter k are the following:

1. Calculate $w = 1 + \lfloor \frac{n}{2^k} \rfloor$.
2. Compute $f(n) = \lfloor \log_2[1 + \frac{n}{2^k}] \rfloor$.

```
if(n==0){
        exp=0;
        len=k+1;
}else{
        //Calculate 'f' and 'w'
        //w=(uintmax_t)(1+floor(n/(1<<k)));
        w=(uintmax_t)(1+(n>>k));
        //f=floor(log2(w))
        f=((8*sizeof(int))-__builtin_clz(w)-1);
        //Calculate the unary code of 'f'
        exp=(((1<<f)-1)<<1);
        //Now follow with the 'f' LSB in binary of w
        unsigned int s2= w & ((1<<f)-1);
        exp=(exp<<f)|s2;
        //Now follow the 'k' LSB of n
        exp=( exp<<k | (n & ((1<<(k))-1)) );
        //Calculate the length
        len=1+2*f+k;
}
```

FIGURE 3.3: Exponential Golomb coder implementation.

3. Construct the code as the unary representation of f followed by the f LSB of the binary representation of w and followed by the k LSB of the binary representation of n.

In the algorithm implementation, the coding of the zero value can be optimized by just writing 0 with $k+1$ bits. If the value is not zero, then we must continue with the coding process. Also, Eq. (3.1), which yields the parameter w, can be implemented as a rightward bit shift by k positions. This procedure allows to obtain the value of $\lfloor \frac{n}{2^k} \rfloor$. We just have to add 1 to compute w.

The straightforward implementation of f would be using the `log` function over w. However, it must be taken into consideration that this operation has a very large computational cost. Therefore, the usage of this function has been replaced by an optimized algorithm. It is important to understand that this parameter corresponds to the leftmost one in the binary representation of w. This helps to develop a computationally efficient implementation of the coder. An extended discussion about how to implement the $\lfloor \log_2 n \rfloor$ operation is available in appendix A.

Once both w and f have been computed, the code can be built. The following operations could be implemented into a single statement. However, in figure 3.3 they are presented as separate instructions.

The unary code of f can be obtained by shifting a '1' bit f positions to the left and subtracting 1 to the result. A zero stop-bit can be added by shifting another position

to the left the resulting value. The next operation is to append the f LSB of w in plain binary code. In order to do this, the meaningless bits of w must be discarded by applying a mask that keeps the f LSB. Another left-shift by f bits, combined with a bit-wise `OR` operation with the truncated w value, will produce the required output. Finally, the last step is to combine the obtained value with the k LSB of x. This is done using equivalent steps as when the LSB of f were added. Finally, the length of the produced code must be computed in order to adequately write the coded value. This can be easily obtained following Eq. 3.3 as f and w are already available.

In order to test the CLDCR with the exponential Golomb algorithm as the coder, we have developed a complete implementation of the CCSDS compressor structure. All the coding options, as well as the mapping method — Prediction Error Mapper (PEM) — have been implemented. The CLDCR has been implemented in `C++` as a modular structure with separate classes for the compressor and the coder. The compressor classes work with blocks of data combined with the adaptive CLDCR stage. The coder class receives the values to be coded plus the configuration parameters and outputs the corresponding result. Additionally, the coder class can compute the length of a code and return this information to the compressor class in order to decide the best coding strategy. This implementation resembles the most typical on-board modular systems. Thus, it helps in obtaining more reliable results. This structure also allows, using class inheritance, a much more flexible compressor implementation. In this case, specific classes for both the exponential Golomb coder and compressor were devised, with the corresponding modifications in order to use exponential Golomb codes within the CLDCR.

3.4 Results with synthetic data

As explained in chapter 3.1, an adequate preprocessing stage leads to prediction errors following a Laplacian distribution. Gaussian distributions are also possible in some cases, although the fact is that the resulting distribution in realistic scenarios is sometimes unknown. Despite of this, the results obtained testing entropy coders on data following Laplacian distributions should be a good hint of the compression performance that we can expect when applied to real data.

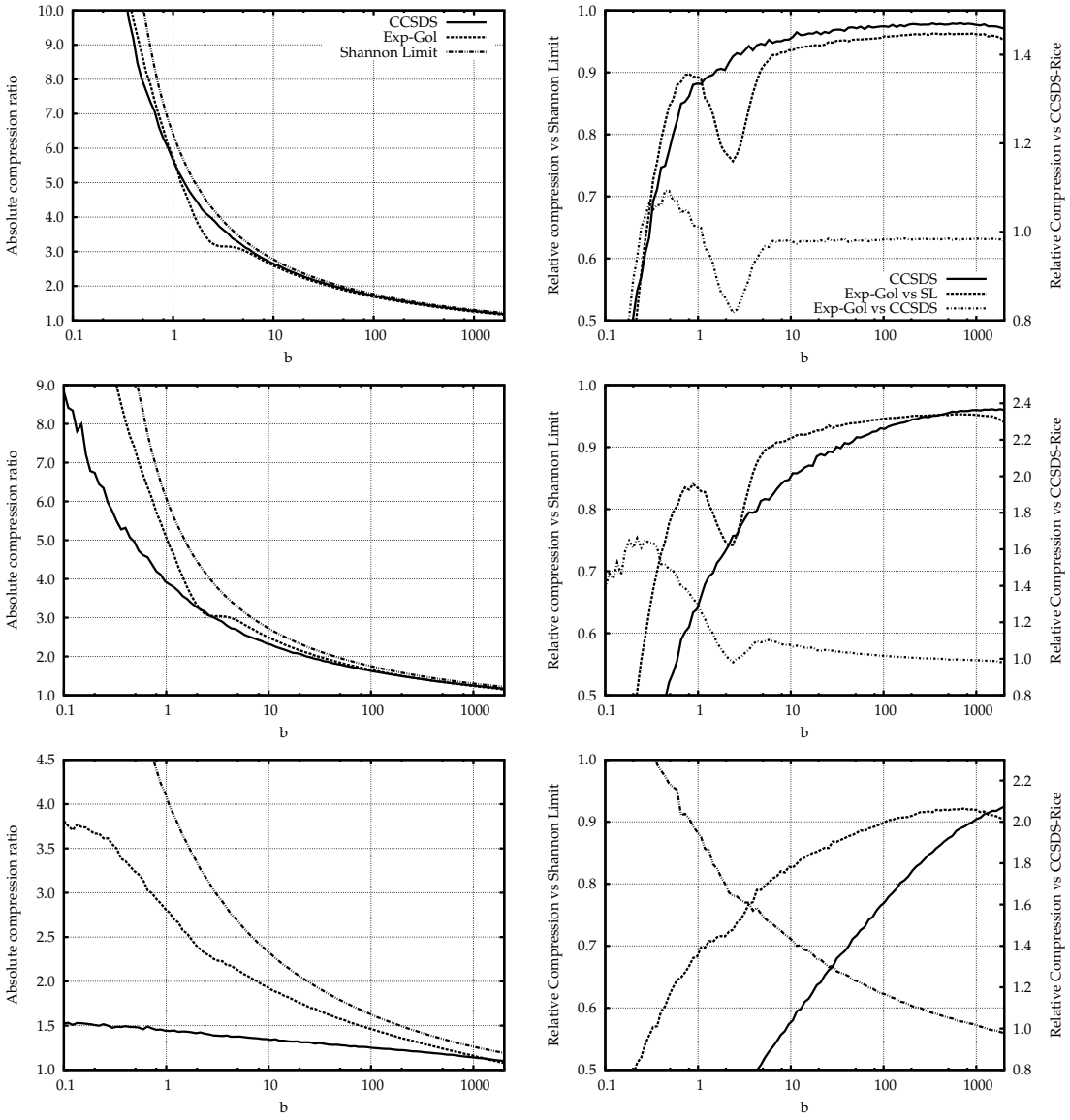

FIGURE 3.4: Compression performance of our adaptive exponential-Golomb coder on synthetic data, for 0.1% (top panels), 1% (middle panels) and 10% (bottom panels) flat noise levels.

Figure 3.4 shows the results obtained when compressing some Laplacian distributions. The panels of this figure cover the entire range of dispersions (or entropy levels) typically found in real cases. The abscissae corresponds to the parameter of the statistic, that is, b for the case of the Laplacian distribution. Small values of b indicate low data dispersion (or, equivalently, low entropy), thus indicating a very good pre-processing stage — or data with implicitly low entropy.

Real data is usually contaminated with noise and outliers. Therefore, to obtain meaningful results the coders have been tested under these conditions. Figure 3.4 presents the

results obtained with different flat noise levels, namely 0.1%, 1% and 10%. These levels represent three different scenarios. An almost ideal scenario where the predictor delivers the expected data distribution corresponds to 0.1% noise level. The more realistic scenario of 1% flat noise offers a view of how the coders perform when 1 of 100 samples is an outlier. Finally, the 10% noise scenario shows the robustness of the compression scheme under extreme situations, a crucial consideration in space applications.

The noise introduced in the samples follows a uniform (flat) distribution in the entire data range. The probability density function for the Laplace distribution is that of Eq. (3.4) where p represents the noise level.

$$f(x) = (1-p)\frac{1}{2b}\exp\left(-\frac{|x|}{b}\right) + p \tag{3.4}$$

Flat noise generally represents the outliers often found in cases in which, for example, CCD samples are contaminated by cosmic rays. It is important to keep in mind the relevance of system stability and tolerance against varying statistics and noise levels. Space instruments usually work in high radiation environments and the mission system has to deal with unexpected behavior of subsystems.

Figure 3.4 shows the performance of the exponential Golomb coder compared to the CLDCR (using Rice) and the Shannon limit. We remind that this is an adaptive coder, owing to the CCSDS 121.0 framework kept in the implementation which selects the best k parameter for each data block — as previously described. From top to bottom, results for 0.1%, 1% and 10% noise levels are shown. The left panels show the absolute compression ratios while the right panels show the relative compression ratios compared to the Shannon limit (left scale) and also against the CCSDS 121.0 standard (right scale).

As commented in section 3.2 the CLDCR compression framework allows the Rice coding strategy to be much more robust in front of noise than what could be expected from a plain (static) Rice implementation. However, when noise is introduced, even the CLDCR decreases its performance rapidly. This behavior can easily be seen by comparing the three sets of figures. Even with moderate noise levels, i.e. 1%, about 15% of the compression ratio is lost in the most common range (from $b = 1$ to 10). Under less favorable scenarios, which nevertheless are not unusual for space applications, the CLDCR

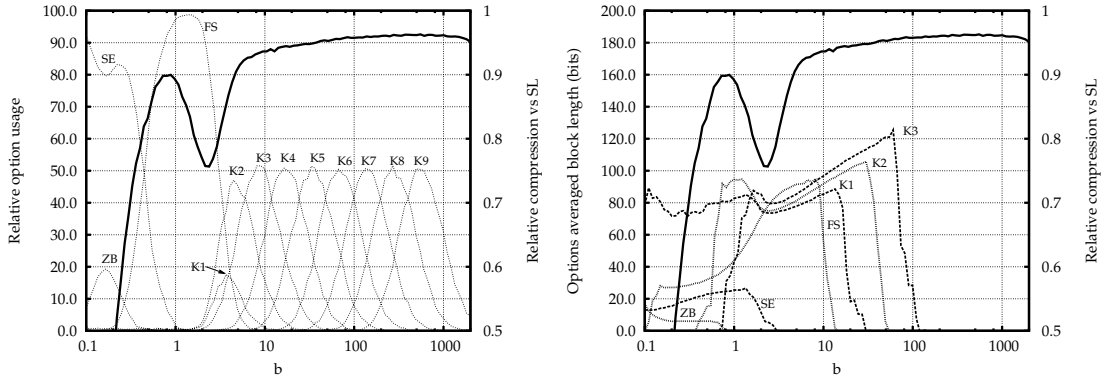

FIGURE 3.5: Relative usage of the compressor options of the exponential-Golomb coder (left) and average compressed block length (right).

is almost unable to provide acceptable compression ratios for any range of entropies. More specifically, ratios of just 1.5 can be obtained in the best of the cases.

When comparing both coders (CCSDS with our Rice implementation and CCSDS with our of the exponential Golomb coder) at low noise levels it can be seen how Rice slightly outperforms the exponential Golomb coder. This is due that in this situation the predictor works properly, thus producing values close to zero. When coding low values with few outliers, low values of k are expected. Rice codes are expected to deliver shorter code lengths and therefore slightly better compression ratios. However, the difference in terms of absolute compression ratios for both coders is almost irrelevant.

As expected, the exponential Golomb coder provides robustness to the compressor architecture when noise is present. However, a critical performance reduction can be observed when b is about 3. A large drop in the compression performance can be observed for both 0.1% and 1% noise levels. This reduction is also present but masked by the general decrease in compression ratios with 10% noise level. To understand this problem, two additional considerations must be taken into account. These are provided in figure 3.5. The left panel of this figure shows the relative usage (or optimality) of each compression option, while the right panel shows the average block length for each compressor option.

The left panel of this figure shows how the different options of the compressor are combined to adapt to the data statistic. The zero block and the second extension options are used with low data dispersions (small b), while the exponential Golomb coder is used for higher values of b. That is an otherwise expected result. On the other hand, it is specially relevant to mention how $k = 1$ and $k = 2$ have a smaller relative usage with

respect to other values of k. Not only that, actually the $k = 0$ option is not used at all. This is due to the fact that they are unable to provide short enough codewords. Therefore, the fundamental sequence coding is used beyond its intended range, and as a result, it produces the severe performance drop seen around $b \simeq 3$. Larger values of k have correct transitions between them, allowing good compression ratios. It is worth mentioning that in this implementation the minimum k value allowed for the exponential coder is 0. The CCSDS 121.0 standard limits the minimum value of k to 1. However, as previously said, even with this parameter option available, the coder fails to deliver short enough codewords, so the $k = 0$ coding option does not even appear in the option usage plot of figure 3.5. The right panel of figure 3.5 displays the average block length and provides another point of view of the same problem. As clearly seen in this figure, the exponential Golomb lengths are larger than those provided by the fundamental sequence where the efficiency drop is found. Fundamental sequence coding was not intended for these data dispersions, hence the poor performance.

As a conclusion, the exponential coder is able to provide robustness to the CCSDS compressor structure against noise and outliers but fails to implement a good transition between the fundamental sequence and the exponential coding. The slightly longer codewords for small values of k have proved to be excessive for this compression architecture. A coder with smoother code length start but able to maintain the exponential growth might better suited. This will be the subject of the next chapter of this report.

3.5 Exponential Golomb decoder

Extensive code revision and testing has been conducted to avoid possible implementation errors. Additionally, a decoder has been implemented in order to guarantee the feasibility of this data compression implementation, thus revealing any possible programming glitch while providing an end-to-end testing (and operational) environment. The decoder has been implemented in a separate executable and using a separate class structure. The class distribution is similar to the one present in the coder and easily extensible to host other decoding algorithms.

It should be noted that, in order to recover the original value, the parameter k used in the coding process must be known. This is something already envisaged in the adaptive

```
if(membuff->GetBit()==0){
        w=(uintmax_t)membuff->Get(k);
}else{
        f=1; //We have read already one 1 bit
        //First read and decode the unary code of 'f'
        while(membuff->GetBit()==1){
        f++;
}

//Now read 'f' bits to recover the 'f' LSB of 'w'.
w=membuff->Get(f);
//Put a '1' after the LSB of 'w'
w=((1<<f)|w);
//Substract '1' and multiply by 2^k to recover 'x' without the 'k' LSB.
w=(w-1)<<(int)k;
//Now read the 'k' next bits, which are the 'k' LSB of 'x' and add them to the result
uintmax_t r=(uintmax_t)membuff->Get(k);
w=w|r;
}
return w;
```

FIGURE 3.6: Exponential Golomb decoder implementation.

framework used (that is, the CCSDS 121.0 framework), which outputs the k used for each compressed data block. Assuming that k is available, exponential Golomb codes generated following the directives specified in section 3.3 can be decoded following these steps:

1. Read the first bit of the coded stream. In case it is zero, read the following k bits and this is the original value. Otherwise, read and decode the unary code of f.

2. Read f bits. These bits will contain the f least significant bits of w. In order to understand the decoding procedure one must remember that, as explained in section 3.3, these are the bits following the leftmost 1 bit in the representation of w.

3. Insert a 1 bit to the left of the f read bits. The result will be the w value as obtained in the coding stage.

4. Subtract one and shift left the result k positions to recover the value of n without k LSB.

5. Finally, read k bits, which correspond to the LSB of n, and add them to the previous value.

The implementation of this algorithm can be simplified by providing a routine to read a specific number of bits from the compressed file.

It is worth mentioning that, in order to recover the unary code of f, '1' bits must be read one-by-one until the '0' stop bit is found. Each '1' value read must be counted, obviously considering the first one already read in the first conditional statement. The final value of this counter corresponds to f.

Figure 3.6 shows an implementation of the decoder algorithm. Using an implementation of this algorithm with the corresponding decompressor routines, compressed files have been restored without any binary difference between original and the restored result.

Chapter 4

Subexponential coder

4.1 Theoretical basis of subexponential codes

We have previously described the motivation of introducing exponential codes in the CCSDS 121.0 compressor structure. As indicated there, when noise or outliers are present in the data, the CLDCR compressor performance quickly degrades. On the other hand, the exponential coder is not exempt of problems either. In this chapter we intend to test another family of prefix codes, namely, the *subexponential* codes [5], with the intention of obtaining the best possible results yet without changing the overall CLDCR implementation. More specifically, we expect to obtain good compression ratios in situations where samples are close to zero (or, in general, with low entropy levels), and at the same time reduce the expansion when outliers or noise are present.

Subexponential codes are used in the Progressive Fast, Efficient, Lossless Image Compression System (FELICS) [5, 15]. Similarly to the Golomb codes, the subexponential coder depends on a configuration parameter k, with $k \geq 0$. Actually, subexponential codes are related to both Rice and exponential Golomb codes. The design of this coder is supposed to provide a much smoother growth of the code lengths, as well as a smoother transition from the inherent CLDCR strategies (ZB, SE or FS) to the prefix coding strategy. In particular, for small dispersions, moving from these strategies to subexponential coding does not imply a significant increase in the output code lengths and, thus, we avoid the poor performance of the exponential Golomb coder in this region.

n	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
0	0|	0|0	0|00	0|000	0|0000	0|00000
1	10|	0|1	0|01	0|001	0|0001	0|00001
2	110|0	10|0	0|10	0|010	0|0010	0|00010
3	110|1	10|1	0|11	0|011	0|0011	0|00011
4	1110|00	110|00	10|00	0|100	0|0100	0|00100
5	1110|01	110|01	10|01	0|101	0|0101	0|00101
6	1110|10	110|10	10|10	0|110	0|0110	0|00110
7	1110|11	110|11	10|11	0|111	0|0111	0|00111
8	11110|000	1110|000	110|000	10|000	0|1000	0|01000
9	11110|001	1110|001	110|001	10|001	0|1001	0|01001
10	11110|010	1110|010	110|010	10|010	0|1010	0|01010
11	11110|011	1110|011	110|011	10|011	0|1011	0|01011
12	11110|100	1110|100	110|100	10|100	0|1100	0|01100
13	11110|101	1110|101	110|101	10|101	0|1101	0|01101
14	11110|110	1110|110	110|110	10|110	0|1110	0|01110
15	11110|111	1110|111	110|111	10|111	0|1111	0|01111
16	111110|0000	11110|0000	1110|0000	110|0000	10|0000	0|10000

TABLE 4.1: Some subexponential codes.

Essentially, subexponential codes are a combination of Rice and exponential Golomb codes. There are two coding strategies depending on the value being coded and the value of k. When $n < 2^{k+1}$, the length of the code increases linearly with n, while for $n \geq 2^{k+1}$ the length increases logarithmically. This first linear part resembles a Rice coding strategy and maintains a slow code growth for small values, while the second part resembles the exponential Golomb code. Table 4.1 shows some subexponential codes for several values of n and k.

These two different coding strategies provide an advantage in front of both Rice and exponential Golomb codes. This definition allows the code to obtain similar code lengths to Rice for small entry values. Additionally, in case of outliers or large values, the code length is shorter than that of Rice due to the exponential steps in the second stage. While this second exponential behavior is also present in the exponential Golomb coder, the average code length is estimated to be shorter, since smaller values obviously will have larger probabilities. Specially, in those scenarios where there are few or no outliers, the coder is expected to deliver higher compression ratios than the exponential Golomb coder while at the same time providing robustness against outliers.

Entering into implementation details, the subexponential algorithm needs two intermediate values which are used in the coding process, namely, b and u. These depend on

the coded value n as can be seen in Eqs. (4.1) and (4.2).

$$b = \begin{cases} k, & \text{if } n < 2^k, \\ \lfloor \log_2 n \rfloor, & \text{if } n \geq 2^k. \end{cases} \tag{4.1}$$

$$u = \begin{cases} 0, & \text{if } n < 2^k, \\ b - k + 1, & \text{if } n \geq 2^k. \end{cases} \tag{4.2}$$

Once these values are obtained, the code can be constructed by coding u in unary and continuing with the b LSB of n. The length of the subexponential code can be extracted from this method. This is done using Eq. 4.3, which is shown for both parts of the code. It can be shown that, for a given n, the code lengths for consecutive k differ by at most 1 bit.

$$l(n) = (u+1) + b = \begin{cases} k+1, & \text{if } n < 2^k, \\ 2\lfloor \log_2 n \rfloor - k + 2, & \text{if } n \geq 2^k \end{cases} \tag{4.3}$$

This coder implementation is detailed in section 4.2 below. Nevertheless, figure 4.1 shows an example of how to code a given value using the subexponential algorithm. In this case, $n = 11$ (1011 in binary) and $k = 2$. The unary coding is shown as n "ones" followed by a zero stop-bit, although ones and zeroes are interchangeable without loss of generality.

$$\begin{cases} k = 2_d \\ n = 11_d = 1011_b \end{cases} \Rightarrow \text{as } n > 2^k \Rightarrow \begin{cases} b = \lfloor \log_2 n \rfloor = \lfloor \log_2 11 \rfloor = 3_d \\ u = b - k + 1 = 2_d \end{cases}$$

$$\underbrace{\underbrace{110_b}_{\text{Unary code of } u + \text{stopbit}} + \underbrace{011_b}_{b \text{ last bits of } n}}_{110011_b = 51_d}$$

FIGURE 4.1: Subexponential coding example

Similarly, as in the case of exponential Golomb codes seen in section 3.2, some tests have been conducted comparing code lengths obtained with compressors based on subexponential and Rice codes. Figure 4.2 reveals the differences in code lengths between Rice and subexponential (left panel), as well as the differences between Rice and the

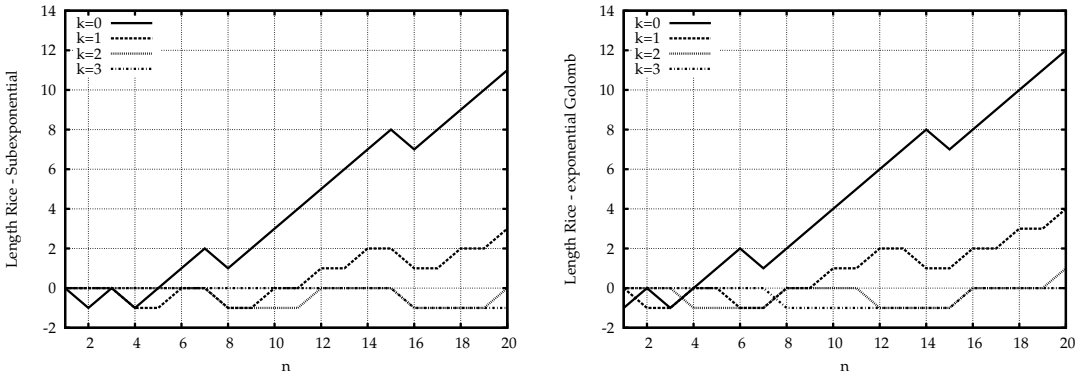

FIGURE 4.2: Code length differences between the Rice, exponential-Golomb and subexponential coders, for small input values.

exponential Golomb coder described in the previous chapter (right panel), which is just a zoom on figure 3.2. We remind that large values in these figures mean better performance than Rice, and vice verse. Both panels are similar, meaning that both coders actually behave similarly for small values. However, if paying attention to both panels we can see what makes the subexponential better than exponential Golomb. Both coders generate codes 1 bit larger than Rice for some values, due to the increase in the b length. Nevertheless, for each given k, subexponential is able to offer the same length as Rice for more values (up to $n < 2^{k+1}$), while the exponential Golomb coder increases its length earlier (at $n < 2^k$). If the predictor works adequately, values close to zero are far more probable than large values, so this advantage of the subexponential coder should lead to better results than those obtained using the exponential Golomb coder. On the other hand, for large values (the most relevant ones in figure 3.2), the results obtained using the subexponential coder are essentially the same obtained using the exponential Golomb coder, so we keep the resiliency in front of outliers. The subexponential coder uses to generate codes 1 bit longer than the exponential Golomb coder for large values of n, but in those regimes the total code length is already large and, hence, the relative effect is much smaller than for small input values.

4.2 Practical implementation of the subexponential coder

One of the requirements for data compression in space is a simple and efficient coding process. The subexponential code of a nonnegative integer n can be computed in three steps, namely:

```
//Calculate 'b' and 'u'
if(n<(1<<k)){
        //If n<2^k
        b=k;
        u=0;
}else{
        //b=floor(log2(n)) u=b-k+1
        b=((8*sizeof(int))-__builtin_clz(n)-1);
        u=b-k+1;
}
subexp=(((((1<<(u))-1)<< 1)<<(b)) | (n & ((1<<(b))-1)));
```

FIGURE 4.3: An optimized C/C++ implementation of the subexponential coder.

1. Compute b and u, as shown in Eqs. (4.1) and (4.2).

2. Code in unary the value of u, here shown as u bits set to '1' and followed by a zero stop bit.

3. Finally, append the b LSB of n to produce the subexponential code of n.

The detailed procedure is as follow. First, we build the unary code of u, which can be obtained by shifting a 1 bit u positions to the left and subtracting 1 to the result. A zero stop-bit can be added by shifting another position to the left the resulting value. The next operation is to append to this value the b LSB of n expressed in plain binary code. In order to do this the non-significant bits of n must be discarded — that is, the all-zero Most Significant Bits (MSB) — by applying a mask that keeps the b LSB. A bit-wise `OR` operation with the truncated value of n will produce the required output. Once b and u are available, the subexponential code can be actually computed. Finally, the length of the produced code must be computed in order to properly transfer the coded value to the next stage. This can be easily obtained following Eq. (4.3) as b and u are already available.

The code definition using b and u requires that the algorithm behaves differently for $n < 2^{k+1}$ than for $n \geq 2^{k+1}$. This is due to the two different coding strategies mentioned before. The implementation of the first strategy is straightforward, as only assignments are required. On the other hand, when $n \geq 2^{k+1}$, the straightforward implementation of b would be using the `log` function over w. However, it must be taken into consideration that this is probably the most computationally expensive operation. Therefore, the usage of this function has been replaced by an optimized algorithm. An extended discussion about how to implement the $\lfloor \log_2 n \rfloor$ operation is available in appendix A.

Tests using the subexponential coder within the CLDCR framework have been conducted, the results of which are shown in the next section. For this, we have reused the complete implementation of the CCSDS 121.0 compressor structure indicated in the previous chapter. All the coding options are thus available, including the prediction error mapping method (PEM). We remind that separate classes are used for the compressor and the coder. The compressor classes work with blocks of data samples combined with the adaptive CLDCR stage. The coder class receives values and parameters to code the values and outputs the corresponding result. Additionally, it can compute the length of a code and return this information to the compressor class in order to decide the best coding strategy.

4.3 Results on synthetic data

Here we discuss the results of the synthetic data tests using the subexponential coder within the CLDCR adaptive framework. Figure 4.4 shows the results obtained from the same tests described in chapter 3.4. That is, random data with a Probability Density Function (PDF) resembling discrete Laplacian (or double-sided geometric) distributions, covering the most typical dispersion ranges and including different levels of flat noise.

Figure 4.4 shows the compression performance of the adaptive subexponential coder, both in terms of absolute ratios (left panels) and also ratios relative to those of the original CLDCR (using Rice) and the Shannon limit (right panels). From top to bottom, 0.1%, 1% and 10% noise levels are displayed.

The first remarkable result that can be seen in figure 4.4 is that the subexponential coder roughly matches or even slightly exceeds the CCSDS performance when very few outliers are present in the data — that is, for the case in which only 0.1% flat noise is added (top panels). Particularly relevant is that for small values of b, which can be rather common in several cases, the subexponential algorithm performs better than the current standard, providing compression ratios which are about 2% larger. In the case of medium to high entropy levels, we are slightly below the CLDCR performance. Fortunately, in this region (where low compression ratios are obtained anyway), the difference is actually negligible.

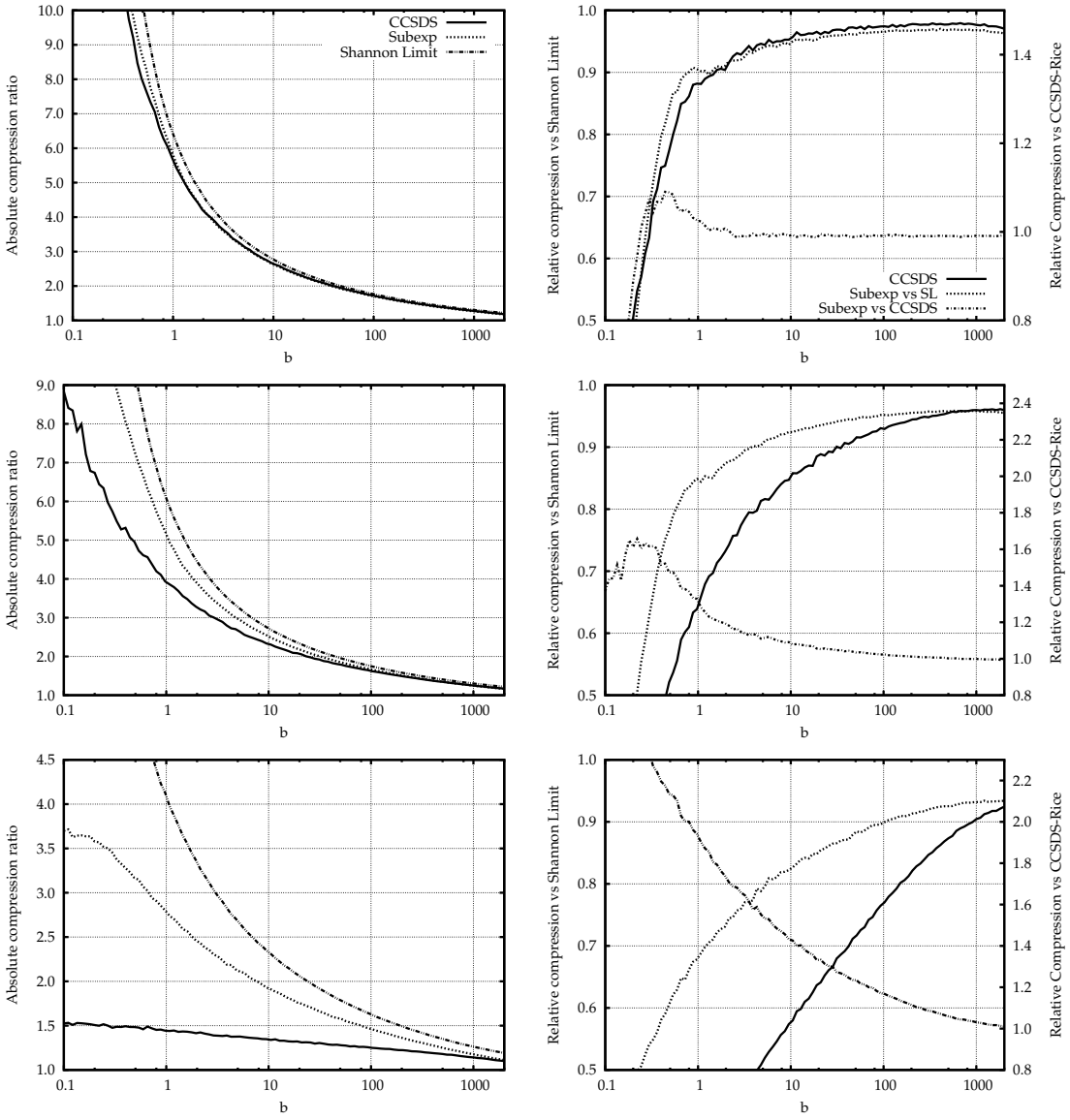

FIGURE 4.4: Compression performance of the adaptive subexponential coder on synthetic data, for 0.1% (top), 1% (center) and 10% (bottom) flat noise levels.

When otherwise realistic noise levels are applied — namely, 1% flat noise — the subexponential coder keeps its compression efficiency mostly unchanged with respect to the 0.1% case, while the CCSDS standard is strongly affected. For the most typical dispersions (say, $b \simeq 1$ to $b \simeq 100$) the efficiency of the current CCSDS standard with respect to the Shannon limit is typically below 90%, and it drops up to just 65%. On the other hand, our adaptive subexponential coder always offers efficiencies above 85% — except for the lowest entropy levels, for which it largely outperforms CLDCR anyway.

Finally, in scenarios where the noise or outliers level is rather high (that is, 10%), the current standard is almost unable to actually compress the data. As already seen in

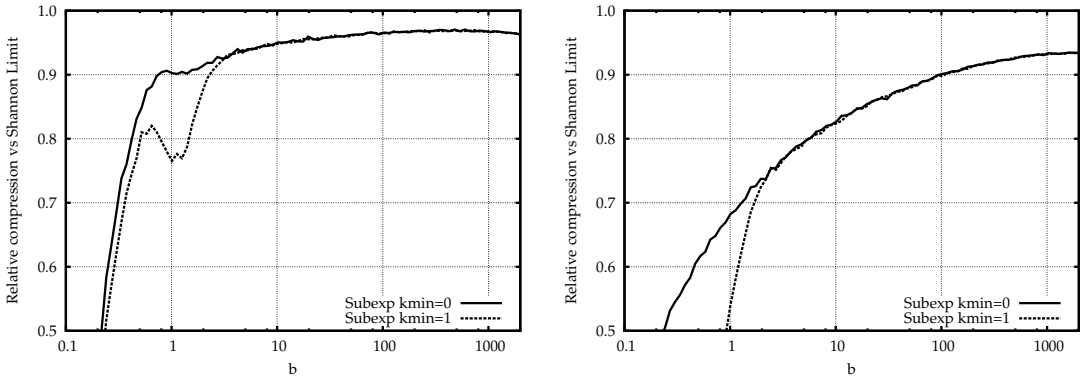

FIGURE 4.5: Compression efficiency of the adaptive subexponential coder with $k_{min} = 0$ and $k_{min} = 1$, for 0.1% (left) and 10% (right) outliers.

the previous chapter, the current CCSDS standard can just reach ratios as low as 1.5 in the best of the cases. That is a compression efficiency well below 50% for medium to low entropies, and typically below 80% even for high entropies. On the other hand, our proposed subexponential algorithm, adequately combined with the CLDCR adaptive framework, obtains compression ratios above 50% for almost any case — even for low entropy levels, while the efficiency is typically above 70%. Ratios up to 3.5 can be reached in this way, which is an excellent result considering the large amount of noise in the data. When compared to the CCSDS standard, our coder can even double the compression ratio under such conditions, while the relative improvement is typically above 1.4.

As we could otherwise expect, the large performance drop observed around $b \simeq 3$ in the case of the exponential Golomb coder has disappeared. Figure 4.5 provides some insight about how this has been achieved. The CLDCR allows the value of k to vary from 1 to 13. In our implementation, it has been modified in order to allow $k = 0$. The largest value of k has been rejected as it was not necessary with the new coder. The behavior of the exponential and subexponential codes rendered irrelevant the $k = 13$ option. In the case of the subexponential coder, its design combined with the use of the $k = 0$ option allows this solution to match and even exceed the CCSDS performance, as we have just seen. Moreover, this modification allows larger compression ratios for low dispersions when the coder is fed with data containing samples with 10% flat noise. Figure 4.5 demonstrates the advantages of allowing the $k = 0$ coding option, while figure 4.6 (specially the right panel) confirms the usefulness of this option in the subexponential

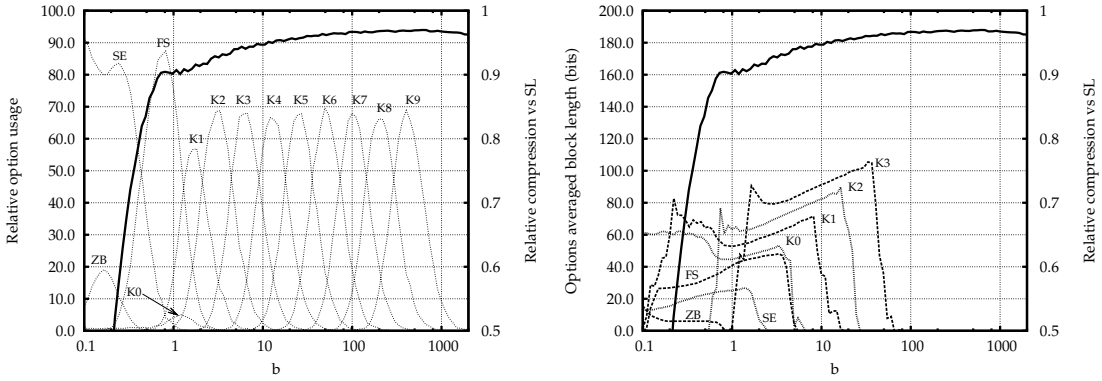

FIGURE 4.6: Relative usage of the subexponential compressor options (left) and average compressed block length (right).

coder. At the same time, it demonstrates that the problem seen with the exponential Golomb coder is due to the uselessness of such $k = 0$ option with that coder.

Now that we are confident on the excellence of this entropy coding solution, we conducted some tests with different sizes of the data compressor blocks in order to check if we can boost further the compression ratios. By default, blocks of 16 samples have been used in our tests. The CLDCR standard allows blocks of either 8 or 16 samples. We suppose that such small sizes were chosen by the CCSDS owing to the high sensitivity of the Rice coder to outliers. Small block sizes probably reduce the effect of such outliers in the original CLDCR. In our case, considering the resiliency of the subexponential coder to outliers, we can safely explore larger block lengths. Using larger data blocks reduces the impact of the block header on the final ratio. Figure 4.7 displays the compression gain when working with 32 samples instead of 16. Particularly, for low entropy levels and in low-noise scenarios, an improvement of up to 5% is achieved. Using data blocks of 32 samples is still safe for space environments [10, 16, 17].

4.4 Subexponential decoder

As it has already been commented in section 3.5, an extensive code revision and testing has been conducted on the coder implementation to avoid possible mistakes. However, the implementation of a decoder was considered mandatory in order to fully guarantee the reliability of our adaptive subexponential coder. The decoder has been implemented in a separate executable, using a separate class structure. The result is an executable

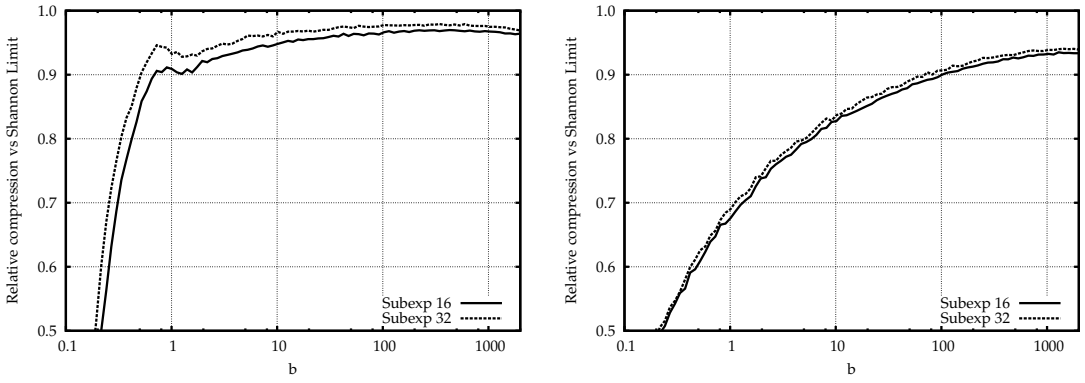

FIGURE 4.7: Subexponential performance with 16 and 32 samples per block for 0.1% (left) and 10% (right) of outliers.

where we can choose either the original CCSDS compressor, the exponential Golomb compressor or the subexponential compressor. In order to decode the values obtained in the coding stage as shown in section 4.1 the parameter k employed must be known. As in both the CLDCR and the exponential Golomb compressors, the subexponential coder also outputs as a small header the k used for each data block. Assuming that the the value of k is known, subexponential codes created following the directives specified in section 4.2 can be decoded following these steps:

1. Read the first bit from the code.
2. If the first bit is 0, the following k bits are the decoded value. Otherwise, the unary code of u must be read.
3. Next, the b LSB have to be read, where $b = u + k - 1$.
4. The last step is to restore the original code by adding to 2^b the b LSBs.

Figure 4.8 shows the implementation of these steps. Simplicity has been a premise but it might be worth mentioning that in order to recover the unary code of u the number of '1' bits until the stop bit '0' must be counted one by one. It must be taken into consideration that the first 1 has already been read. The resulting count will correspond to the parameter u.

After implementing this algorithm within the adequate decompressor routines, compressed files have been successfully restored. By comparing them with the original files we

```
uintmax_t result=0;

//Subexponential has two different cases
if((membuff->GetBit())==0){
        //Case n<(2^k)
        return (int) membuff->Get(k);
}else{
        //Case n>(2^k)
        int b,u;
        uintmax_t base;
        //Count the number of 1's until the stop bit.
        u=1; //We already have read the first 1
        while(membuff->GetBit()){
                u++;
        }
        //Now the least b significant bits have to be read.
        b=u+k-1;
        base=1<<b;
        result = (base | membuff->Get(b));
        return (int)result;
}
```

FIGURE 4.8: Subexponential decoder implementation

have assessed that they are identical up to the last bit, so we have confirmed that the adaptive subexponential compressor is indeed lossless.

Chapter 5

REGLIUS and HyPER Coder

5.1 Interest in hybrid PEC/Rice coding

In the search for an efficient and resilient entropy coder, in chapters 3 and 4 we have discussed two modifications to the CCSDS 121.0 standard. In both cases, our intention was to obtain the best possible results with the minimum modifications to a well-known and reliable compression system. Nevertheless, other compression strategies should also be investigated, even if implying radical changes in the design.

The PEC and FAPEC coders described in chapter 2 are good examples of excellent entropy coders requiring a completely different strategy than that of Rice or the CCSDS 121.0 recommendation. Several tests on these systems, which can be found in [10], demonstrate that they outperform the CCSDS 121.0 standard in most of the realistic scenarios. The segmentation strategy designed in PEC delivers outstanding results when noise or outliers are present. On the other hand, the tests presented in chapter 4 reveal that the adaptive subexponential coder, making use of the CCSDS 121.0 framework, also has an excellent behavior under noisy scenarios, while the penalty for low entropies and clean data is often smaller than in PEC or FAPEC. Thus, on one hand there is the PEC-based segmentation strategy which appears to be excellent for very noisy environments. On the other hand, the Rice-based coding offers a smoother increase in the code lengths that benefits clean environments and small entropies. Thus, it is rather obvious that a combined strategy should deliver excellent results.

In this chapter we explore the idea of combining these two strategies into a single entropy coder, that is, a hybrid between Golomb codes and PEC-based coding. First of all, we define here a new code which will be used as the base of a segmented coding strategy similar to that of PEC. The idea is to use this code instead of the plain binary coding used in each of the PEC segments, looking for a smoother increase of the code length. We have called it REGLIUS, and resembles the subexponential coding in the sense that it combines the Rice-based coding for the smallest values with the exponential Golomb b increases for larger values. Nevertheless, it has a limited coding range as it will be seen below. This limitation is introduced in order to ease the implementation and calibration of the segmented coder, also discussed later. The segmented coding strategy has been called HyPER coding, since it combines the PEC-based segmentation with the REGLIUS codes in each segment. The HyPER coding strategy is expected to be robust against noise and outliers while yielding excellent results for clean data and small entropies, thus outperforming the current standards. Let us describe REGLIUS and the HyPER coder in the following sections.

5.2 The REGLIUS codes

One of the main properties of REGLIUS codes is their limited coding range. In this sense, the strategy of REGLIUS resembles that of a plain binary coding using a given number of bits, rather than a Golomb coding — for which the coding range is infinity. This limitation has been imposed, on one hand, to simplify the implementation and calibration of a segmented coding strategy similar to PEC or FAPEC. On the other hand, it allows to make use of all of the bits available in the Rice-Golomb definition for actual value coding, including the stop bit.

Another feature of the REGLIUS codes is that they depend on a $k \geq 0$ parameter, in a similar manner as Golomb codes do. We have designed the code in such a way that the maximum value that can be coded with a given configuration is easily computable. More specifically, a REGLIUS code with a given k configuration is able to code from zero up to $2^{k+3} - 1$. Thus, it is equivalent to a plain binary code of $k + 3$ bits. The difference is that the size of this code spans from 2^{k+1} bits up to 2^{k+6} bits.

n	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
0	0|	0|0	0|00	0|000	0|0000	0|00000
1	10|	0|1	0|01	0|001	0|0001	0|00001
2	110|	10|0	0|10	0|010	0|0010	0|00010
3	1110|0	10|1	0|11	0|011	0|0011	0|00011
4	1110|1	110|0	10|00	0|100	0|0100	0|00100
5	1111|0	110|1	10|01	0|101	0|0101	0|00101
6	1111|10	1110|00	10|10	0|110	0|0110	0|00110
7	1111|11	1110|01	10|11	0|111	0|0111	0|00111
8	N/A	1110|10	110|00	10|000	0|1000	0|01000
9	N/A	1110|11	110|01	10|001	0|1001	0|01001
10	N/A	1111|00	110|10	10|010	0|1010	0|01010
11	N/A	1111|01	110|11	10|011	0|1011	0|01011
12	N/A	1111|100	1110|000	10|100	0|1100	0|01100
13	N/A	1111|101	1110|001	10|101	0|1101	0|01101
14	N/A	1111|110	1110|010	10|110	0|1110	0|01110
15	N/A	1111|111	1110|011	10|111	0|1111	0|01111
16	N/A	N/A	1110|100	110|000	10|0000	0|10000

TABLE 5.1: Some REGLIUS codes, for k up to 5 and n up to 16.

REGLIUS codes have four different coding stages. First, they have been defined with a slow growth in their length for small data values — related to low entropy samples. This is achieved with a Rice coding for the smallest values. With this, we should be able to take full advantage of an adequate pre-processing stage. Two Rice *jumps* are allowed, that is, adding up to two bits in the fundamental sequence. When the input value becomes larger, the code closely resembles to an exponential Golomb on — similar to what happens in the case of subexponential codes. One *extension* of the b bits is allowed — that is, allowing to increase the useful coding range just once. When the coding range is exhausted, the stop bit is switched and the remaining bits except one are used. Finally, for the largest possible values of a given code range, we add a final extension of 1 bit to accommodate more values up to $2^{k+3}-1$. Table 5.1 illustrates some REGLIUS codes for $k=0$ to $k=5$ for the 17 first n values. From the coding process shown in this table we can infer the code length, which depends on both k and n (as we could expect) following Eq. (5.1). Figure 5.1 shows an example of the coding process using as an example the value $n=21$ and $k=2$.

$$\text{length} = \begin{cases} k+1+\lfloor \frac{n}{2^k} \rfloor, & \text{if } n < 3\cdot 2^k, \\ k+5, & \text{if } 3*2^k \leq n < 3\cdot 2^{k+1}, \\ k+6, & \text{if } 3*2^{k+1} \leq n < 2^{k+3}. \end{cases} \tag{5.1}$$

We can thus compare REGLIUS codes against their equivalent binary coding (that is, using $k+3$ bits), which is used in the PEC segments. REGLIUS codes are up to 2 bits shorter than standard binary for the first 2^{k+1} values. The next 2^k values require exactly the same length than in standard binary coding. And finally, the remaining values are coded with up to 3 bits more. That is obviously the penalty for obtaining shorter codes for smaller values. It remains to demonstrate whether this penalty is compensated by the improvement achieved for small values, which should be the case considering the Laplacian probability distributions typically obtained after an adequate pre-processing stage.

$$\begin{cases} k = 2_d \\ n = 21_d = 10101_b \end{cases} \Rightarrow \begin{cases} \text{1st range: } n \leq 11 \\ \text{2nd range: } 11 < n \leq 19 \\ \text{3rd range: } 19 < n \leq 23 \\ \text{4th range: } 23 < n \leq 31 \end{cases}$$

$$\underbrace{\underbrace{\text{Range prefix}}_{11110_b} + \underbrace{n - 5 \cdot 2^2 \text{ in } k \text{ bits}}_{01_b}}_{1111001_b = 121_d}$$

FIGURE 5.1: REGLIUS coding example

5.3 Theoretical basis of the HyPER coder

REGLIUS codes are used in a segmented coding strategy, which is the HyPER coder. As previously noted, this coding strategy has proved to deliver excellent results in both PEC and FAPEC. We will continue using four ranges as in those coders, and we will also continue using signed values — so that no PEM stage is necessary. Each of the four segments uses a different k parameter for its REGLIUS code, which we will name $k1$, $k2$, $k3$ and $k4$. Similarly as in the LE strategy of PEC, the second, third and fourth segment are signaled with the reserved -0 value coded in the first segment. As it can be imagined, REGLIUS will deliver a shorter code for -0 than standard binary, which should lead to an improvement in the compression ratio. The HyPER coder shall also be calibrated using four parameters, as in the case of PEC. In the current implementation, these are static values determined by means of a trial-and-error process, although an adaptive stage might be added in future developments. Figure 5.2 shows the definition

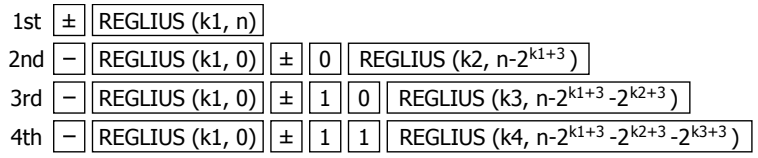

FIGURE 5.2: Implementation of the HyPER coder with four segments.

of this hybrid coding strategy using four segments, where REGLIUS$(k1, n)$ indicates the REGLIUS code of value n using $k = k1$.

5.4 Practical implementation of the HyPER coder

This section describes the practical implementation of both the REGLIUS coding method and of the HyPER coder. As the four segment strategy present in the HyPER coder is build upon REGLIUS codes, these will be explained first.

In REGLIUS coding, the first two coding steps follow a typical Rice coding scheme, and therefore any $n \leq (3 \cdot 2^k - 1)$ will be coded using a standard Rice algorithm. In section 2.1.2 Rice coding is detailed. The next range will add one more useful bit after the Rice code of $3 \cdot 2^k$, leading to $k + 1$ useful bits, and consequently being able to code 2^{k+1} values. This leads to a maximum value of $n \leq (6 \cdot 2^k - 1)$ for this range. For larger input values, the $k + 1$ bits after the fundamental sequence are exhausted. Now the stop bit is going to be reused, allowing 2^k more values to be coded. This leads to a maximum coding value of $n \leq (7 \cdot 2^k - 1)$. The final stage adds yet another useful bit, allowing an additional coding range of $n \leq (2^{k+3} - 1)$.

As previously discussed, REGLIUS has been designed in order to simplify and optimize the coder implementation. Therefore, codes that fall in the second range will be constructed as the fundamental sequence of 3 followed by $n - 3 \cdot 2^k$ coded using $k + 1$ bits in plain binary. The fundamental sequence of 3 is 1110_b, as seen in table 5.1. Once the second range is exhausted, the third range will generate the fundamental sequence of 4 (11110) followed by $n - 5 \cdot 2^k$ coded in k bits, again in plain binary code. Finally, the last range of codes will always be built as 11111_b followed by $n - 3 \cdot 2^{k+1}$ in $k + 1$ plain bits. These are useful properties to optimize the implementation of the REGLIUS

```
//First stage: x<(3*2^k)
if(x<(3*(1<<k))){
        //Rice coding
        z=(x>>k); //z=x/(2^k)
        size=1+k+z;

        if (z!=0){
                code=(((1<<z)-1) << k+1)|(x & ((1<<k)-1));
        }else{
                code=(x & ((1<<k)-1));
        }
}else if (x<(5*(1<<k))){
        //Second stage: x<(5*2^k)
        size=5+k;
        //1110b=14d is always the prefix.
        code=( (14<<(k+1)) | (x-3*(1<<(k))) );
}else if (x<(3*(1<<(k+1)))){
        //Third stage: x<(3*2^(k+1))
        size=5+k;
        //11110b=30d is always the prefix
        code=( (30<<k) | (x-5*(1<<k)) );
}else{
        //Fourth stage: x<(2^(k+3))
        size=6+k;
        //11111b=31d is always the prefix
        code=( (31<<(k+1)) | (x-3*(1<<(k+1))) );
}
return code;
```

FIGURE 5.3: REGLIUS coder implementation in C.

coder, reducing the coding process to the concatenation of a binary value with a constant prefix. Figure 5.3 presents an implementation of the REGLIUS coder.

The HyPER coder structure has four different segments, as shown in figure 5.2. Again, the compression routine has been designed with simplicity in mind. The first segment will be coded with a REGLIUS coding with the *k1* parameter plus the sign bit preceding the code. The following segments will be preceded by the REGLIUS code of the leap value -0 coded with *k1*. The sign bit and a prefix to identify the segment follow. More specifically, 0 for the second segment, for 10 the third and 11 for the fourth. After this, each segment appends a REGLIUS code using the corresponding k parameter. It must be noted that for the second, third, and fourth segments this last part will code the difference of n with respect to the maximum value that can be coded with the previous segment.

5.5 Results on synthetic data

In this section we show the results of the synthetic data tests conducted using the HyPER coder. The panels of figure 5.4 show the results obtained on the same test bench

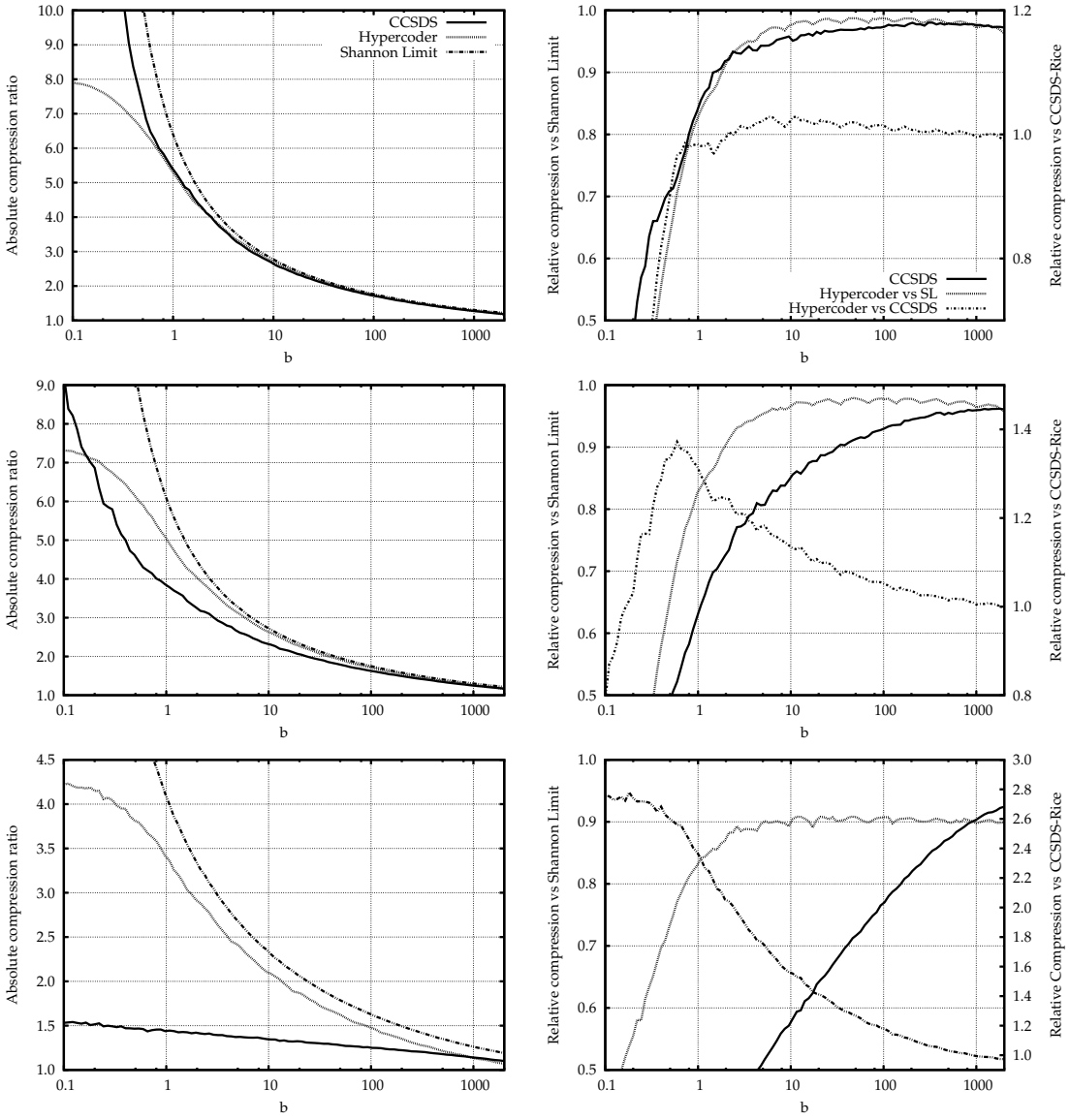

FIGURE 5.4: Compression performance of the HyPER coder versus the CCSDS 121.0 recommendation for 0.1%,1% and 10% flat noise levels.

described in section 3.4. That is, random data with a PDF resembling a discrete Laplacian (or double-sided geometric) distribution, covering the most typical dispersion ranges and including different flat noise levels. The figure shows the compression performance of the HyPER coder, both in terms of absolute ratios (left panels) and also ratios relative to those of the original CLDCR (using Rice) and the Shannon limit (right panels). From top to bottom, 0.1%, 1% and 10% noise levels are applied.

The first remarkable result that can be seen in figure 5.4 is that the performance of the HyPER coder is comparable to that of the current CCSDS standard in low-noise scenarios — namely 0.1% noise (top panels). The HyPER coder developed in this project

slightly outperforms the current standard for typical dispersions ($b = 3$ to $b = 1000$). However, for low dispersions ($b < 3$) the performance decreases when compared to that of the CLDCR. This effect is most probably caused by the separate coding of the sign bit in the current HyPER implementation, which limits the maximum achievable ratio to 8 (in this 16-bit implementation). Additionally, the non-adaptive operation of this HyPER implementation probably affects the final result as well. Finally, an adequate strategy for coding very low entropy levels is not available yet, while the CCSDS 121.0 standard (and also FAPEC) features that. Future developments of this coder should address this issue and add a different coding strategy for this region.

As previously seen in sections 3.4 and 4.3, when otherwise realistic noise levels are applied, the performance of the CCSDS standard abruptly drops. However, the HyPER coder maintains its efficiency levels — and even the absolute ratios. As figures 5.4 illustrate, $b \approx 0.7$ presents a maximum improvement with respect to the CLDCR. For very low data entropies, the HyPER coder offers a slightly worse performance than the standard, while for higher entropies it slowly converges with the standard.

Finally, it is worth highlighting how the designed coder tolerates high noise levels without problems. Even at 10% noise levels, the coder efficiency for relevant dispersion levels ($b = 1$ to $b = 100$) remains at about 90%, clearly outperforming the current standard. Only at very high entropies the CLDCR offers performances similar to those of the HyPER coder, although in this region the compression ratios are anyway too small.

5.6 HyPER decoder

In this section the REGLIUS and HyPER coder decoders implementation done in this work will be discussed. As seen in section 5.3, REGLIUS codes depend on a parameter $k \geq 0$. The parameter used in the coding stage must be known in order to recover the value correctly. In the current design of the HyPER coder the four k parameters have to be calibrated previously. Therefore, in the following process it is assumed that the k value is known.

The REGLIUS decoding process is shown in figure 5.5. The basic idea behind the decoder is to fit the current code in the corresponding range in order to process it accordingly. This is done by reading a single bit in each step until the correct range

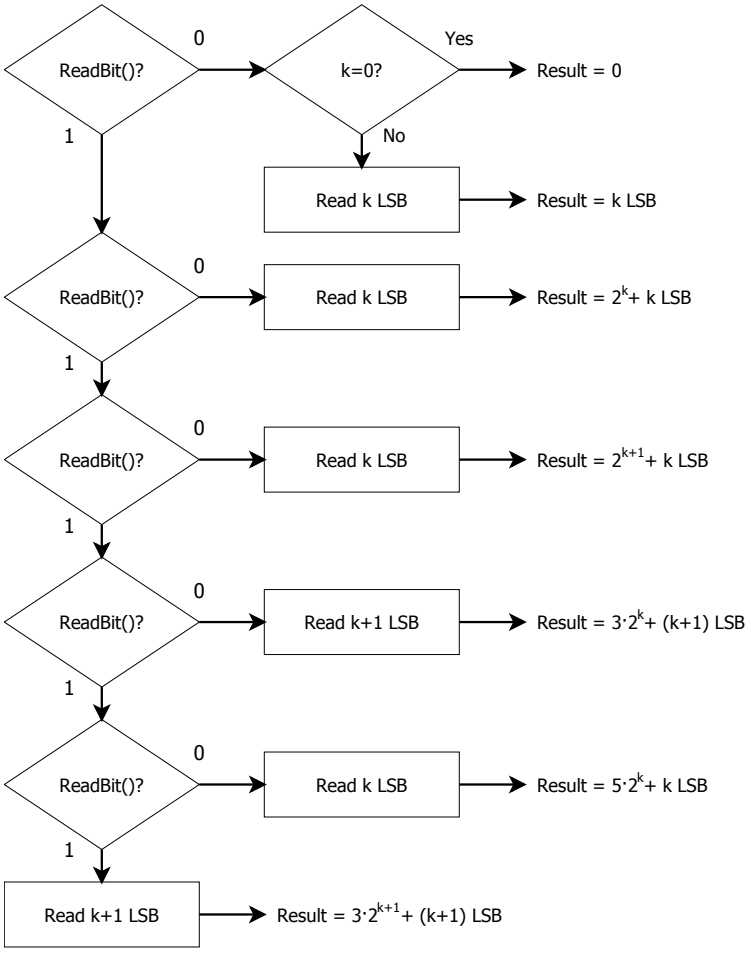

FIGURE 5.5: REGLIUS decoding process.

is found. Once the range is known, the corresponding LSB are read and the result is computed.

This decoding structure is, by design, slower than those corresponding to Rice, exponential Golomb or subexponential codes. However, the decoder is not under the same constraints as the coder as it is expected that it will run on ground. Therefore, this algorithm design does not have such strict power and processing limitations as those on board a satellite.

The implementation of the HyPER coder decoder is straightforward once a routine to decode the REGLIUS codes is available. This section assumes the decoder algorithm previously explained has it implemented. The algorithm to decode a value coded with HyPER must follow these steps:

1. First read the sign bit, which will always be the first one in any of the four segments.

2. Once the first bit is read, a REGLIUS coded value with the parameter *k1* can be read. This is assumed to be decoded using the routines previously explained.

3. With the sign and the decoded value for the first segment it can be determined whether this code belongs to the first segment. This is done by comparing the value with the leap value -0. If the value found is not the leap value, the read value is the result and the decoding stage is over for this code.

4. If the decoded value was -0, the actual sign of the value must be recovered by reading a single bit.

5. Next, the segment identifier must be read. This is done by reading a single bit. If zero, the value was coded using the second segment. Otherwise another bit must be read to distinguish whether it corresponds to the third or the fourth segment.

6. Once the segment is known, the REGLIUS decoding routine can be called again using the corresponding k value.

7. Finally, the maximum value of the previous segment must be added to the decoded value.

The HyPER decoder has been successfully implemented and tested, assessing that this is, indeed, a lossless compression strategy.

Chapter 6

Results

6.1 Results on synthetic data

In this chapter global results for the coders developed in this project and presented in chapters 3, 4 and 5 are compared to the current CCSDS standard and the result of this comparison is discussed. This first section covers a comparison using synthetic data following the same approach described in previous chapters.

Figure 6.1 shows the performance of these coders compared to the CLDCR (using Rice) and the Shannon limit. It should be noted that the CLDCR, the exponential Golomb and the subexponential coders have adaptive stages. This adaptive stage selects the best k parameter for each data block. However, the current implementation of HyPER coder lacks this adaptive stage. Thus, for a meaningful comparison this coder has been calibrated for each data point. From top to bottom, results for 0.1%, 1% and 10% noise levels are shown. The left panels show the absolute compression ratios while the right ones show the relative compression ratios with respect to the Shannon limit.

The first important result that figure 6.1 demonstrates is that for low-noise scenarios the performance of the exponential Golomb coder drops abruptly for $b \sim 3$, as previously discussed. A detailed analysis of this performance drop has been offered in section 3.4. Basically, the exponential Golomb coder is unable to deliver short enough codewords for small values. Other compression strategies present in the CCSDS such as fundamental sequence are not designed to work well in these ranges thus delivering poor results. On the other hand, as discussed in section 4.3, the subexponential coder does not show

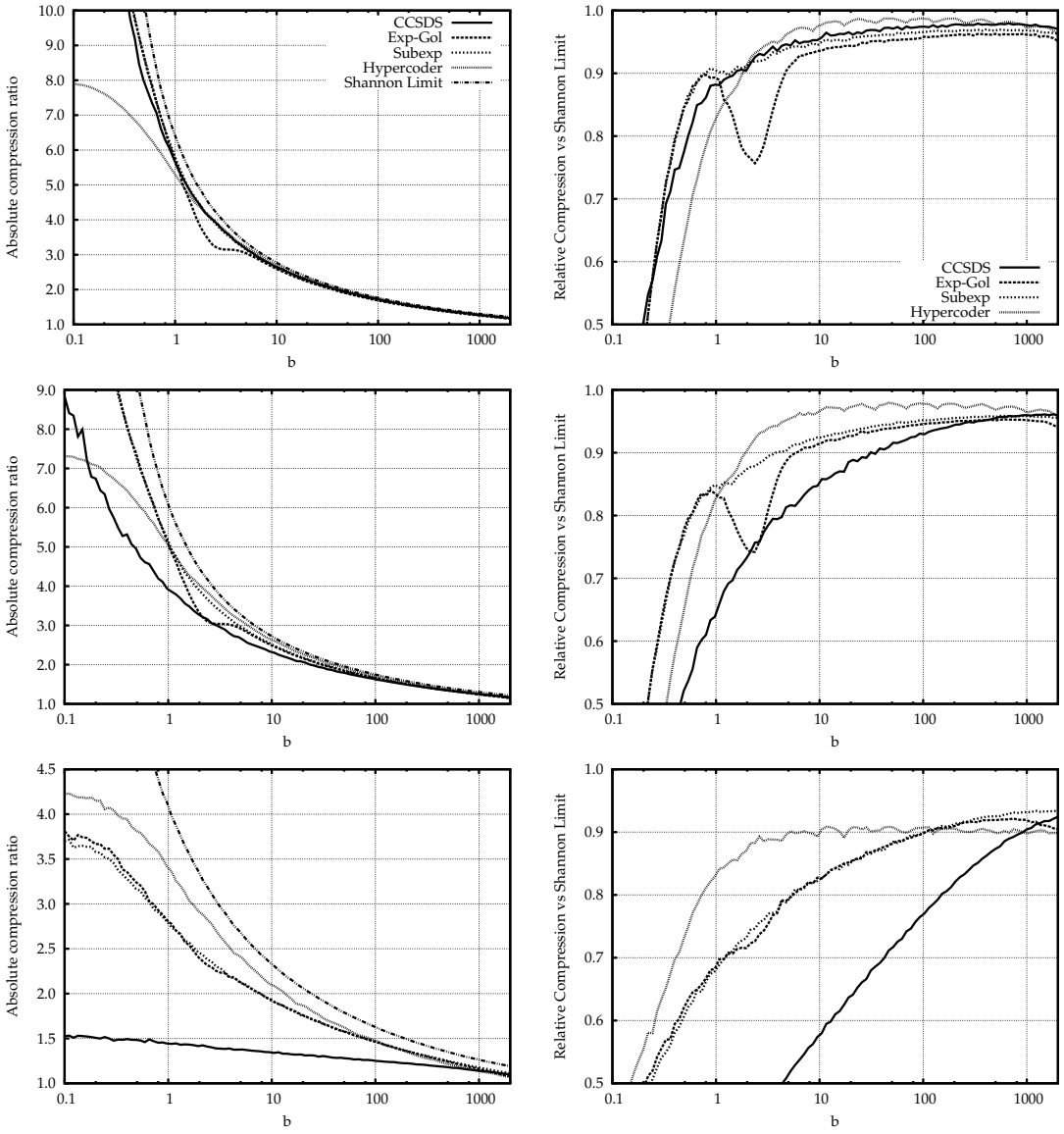

FIGURE 6.1: Performance of the Rice, exponential-Golomb, subexponential coders and of the HyPER coder for 0.1% (top), 1% (middle) and 10%(bottom) flat noise levels.

this large performance drop. This coder has been chosen to provide a smoother code start in terms of code length. This allows the codes for low values to be shorter while maintaining the exponential growth for large values. Both exponential Golomb and subexponential coders outperform CCSDS using Rice when b is between $\simeq 0.2$ and $\simeq 1$. However, for larger entropies the current standard has marginally better results. It must be emphasized as well that the HyPER coder provides the best compression ratio for high entropy levels. This fact demonstrates the robustness of the proposed segment coding approach. However, when $b < 1$ its performance decreases delivering 10% to 15% worse results than the current standard. All the other compression solutions have

different strategies to deal with these low entropies. It was out of the scope of this project to implement such option but next versions of this coder should definitely address this issue.

When considering realistic noise levels, namely 1% noise, the HyPER coder has the best response, yielding the highest compression ratio except for low entropies. Compression gains in the most relevant zone, b from 1 to 100, range from about 15% to 5% compared with the CCSDS strategy using the Rice coder. The developed subexponential coder, even using the same compressor architecture as the CCSDS 121.0 standard, offers gains in the same zone that range from about 15% to 2%. The proposed subexponential coder brings remarkable robustness with minimal changes in the current CCSDS compressor structure.

Finally, the results obtained when considering high noise levels highlight the advantages of the devised segment coding strategy in front of the current CLDCR architecture. The panels in which 10% noise level has been considered clearly show that the current CCSDS standard using Rice is completely unsuitable for noisy environments. Both the exponential Golomb and subexponential coders increase the architecture resilience against noise. Compression levels are increased by more than 20% in both cases. However, when comparing these results with those of the HyPER coder it is clear that this last coder improves further the compression ratio for the region of interest.

6.2 Results on real data

In chapters 3 to 5, the results of the theoretical analysis performed in this work were shown. Each coder was tested under realistic conditions using synthetic data and a comparison between all coders has been presented in section 6.1. However, the final assessment can only be achieved by testing the systems with real data. A compression corpus with real data sources has been used to perform a complete analysis and to offer even more representative evaluations of the performances of the coders. This corpus is a set of data files covering the most typical scenarios which the compression algorithms might face.

All the compression ratios have been computed considering only the information processed and compressed by the coder. In the case of an ASCII input, the binary equivalent

has been used to calculate the initial size. Moreover, for files containing headers, these have been withdrawn, allowing a reliable performance evaluation. More specifically, for all practical purposes headers are not considered by the coders. Finally, it is worth mentioning that additional procedures, such as different pre-processing stages or interleaving, are usually included in the most typical on-board compressor structures. In particular, on-board implementations of the compressors are tailored to the needs of the different missions, and more complex pre-processing stages than the ones used in these tests are also considered as well. Thus, the ratios shown in the following subsections can be considered a worst-case scenario.

As seen in chapter 5 the current HyPER coder implementation only supports 16 bit words. Therefore, it should be noted that HyPER coder ratios are only shown for files where coding with 16 bits codewords makes sense, that is 8, 16, 32 or 64 bits. Furthermore, the HyPER coder was calibrated for each file with the best set of parameters before the compression. Therefore, the HyPER coder compression ratios must be considered as the best result obtained with the coder calibrated specifically for each file. All the other compressors share the adaptive stage present in the CLDCR. An adaptive stage for HyPER similar to the one present in the CCSDS architecture would be unreasonable in a space environment considering the processing limitations. In order to maintain coding speed and limit processing requirements such an implementation should be avoided. A different approach for an adaptive stage which maintains coding speed is present in FAPEC, but might lead to a slight reduction of the compression ratios presented here.

6.2.1 Corpus description

A detailed description of the files used to evaluate the different coders is presented in the following subsections. Data files are grouped according to the type of information and features. Additionally, tables comparing the compression ratios obtained by each algorithm are included. Furthermore, the Shannon limit and the word length for each file are appended in these tables. The best compression ratio achieved is highlighted in blue and it must be noted that results have been rounded to two decimal places. In the following tables some compression values are higher than the corresponding Shannon limit. This is due to the fact that the ZB option present in the CLDCR framework

codes blocks of zeros as simple codewords while the Shannon limit assumes independent events.

6.2.1.1 Images

When considering data compression in space applications, imaging data is probably the most relevant case. Most space missions include cameras, although not necessarily operating in the visual range of the spectrum. Therefore, a variety of space-related images stored in Flexible Image Transport System (FITS) format have been evaluated with the proposed compressors. FITS file format is widely spread since it is specifically devised for the scientific applications. In these files, information is stored as raw (uncompressed) binary data, allowing the data to be easily accessed and visualized. Table 6.1 presents the compression results for the image data set.

Data from the Fiber-Optics Communications for Aerospace Systems (FOCAS) is a standardised data set of astronomical data. This data set is usually used to test calibration methods which should deal with very different astronomical images [18]. Files `gal0003`, `gal0004`, `sgp0001` and `sgp0002` contain 32-bit samples, therefore the sample splitting process has been used.

Gaia Instrument and Basic Image Simulator (GIBIS) images are a subset from the GIBIS group of files described in subsection 6.2.1.2. These files were generated using the GIBIS simulator [19] with highly realistic simulations. These include cosmic rays effects, noise, pixel saturation, and can be considered an excellent approximation to real captured images in space missions.

The miscellaneous group includes a series of astronomical images related to extended sources, such as galaxies, stellar fields, or nebulae. This group differs from the previous two groups in a significant way. Most data in both FOCAS and GIBIS images is background and noise with just some point-like sources corresponding to stars. However in these files astronomical sources cover large areas while few data corresponds to background noise.

Files `ground_1.pgm` and `ground_2.pgm` are stored in Portable Gray Map (PGM) format. These two files contain micro-photographs of ground samples. Hence, they represent images that might be taken by a robotic mission.

File	Size (bytes)	Compression Ratio				S_L	W_L (bits)
		Rice	Gol-Exp	Subexp	HC		
FOCAS							
tuc0003	210240	**2.89**	2.66	2.86	2.84	3.06	16
tuc0004	210240	**3.04**	2.72	3.01	3.00	3.22	16
com0001	504000	1.98	1.91	1.95	**2.01**	2.07	16
ngc0001	331200	1.76	**1.84**	**1.84**	1.75	1.92	16
ngc0002	331200	2.50	2.51	**2.60**	**2.60**	2.75	16
for0001	316800	**3.05**	2.84	2.99	2.06	3.22	16
for0002	308160	**3.98**	3.53	3.89	3.88	4.28	16
gal0001	325440	2.59	2.47	2.53	**2.63**	2.77	16
gal0002	230400	2.17	2.10	2.14	**2.19**	2.30	16
gal0003	4003200	3.58	3.66	3.74	**4.08**	4.98	32
gal0004	4003200	3.53	3.66	3.68	**4.03**	4.95	32
sgp0001	417600	3.34	3.57	3.54	**3.80**	4.77	32
sgp0002	417600	4.04	3.71	4.27	**4.42**	5.45	32
GIBIS							
simu7135_SM1_6	3006721	3.45	3.06	3.39	**3.48**	3.78	16
simu5291_SM1_4	10008001	3.54	3.09	3.46	**3.55**	3.88	16
simu7135_AF1_6	24007681	1.05	1.06	1.07	**1.40**	1.49	32
simu5291_AF1_5	80015041	2.12	2.27	2.22	**2.55**	2.84	32
simu7135_RP1_6	24007681	1.02	1.03	1.04	**1.42**	1.52	32
simu5291_RP1_2	80015041	2.29	2.42	2.37	**2.66**	3.05	32
simu7135_RVS1_6	24007681	1.11	1.14	1.14	**1.54**	1.65	32
simu5291_RVS1_5	80015041	1.94	2.09	2.02	**2.38**	2.70	32
Miscellaneous							
stellar_field_little	734400	1.10	2.22	2.14	**2.72**	5.15	64
accretion_disk	63360	0.81	0.81	**0.82**	0.63	1.30	32
noisy_source	4219200	1.01	1.00	1.03	**1.05**	1.19	32
galaxy	734400	1.14	2.23	2.17	**2.82**	4.85	64
m35_stellar_field	11327040	**1.57**	**1.57**	1.54	1.55	3.34	16
nebula_stellar	731520	1.00	1.19	1.19	**1.67**	3.78	64
ground_1	11039273	**1.38**	1.27	1.35	1.13	1.44	8
ground_2	26214456	**1.92**	1.75	1.86	1.22	1.81	8

TABLE 6.1: Results obtained for image files, classified into three groups depending on the data generator.

6.2.1.2 GIBIS

GIBIS files correspond to highly realistic simulations of Gaia [19]. These files follow the same data format that will be used in the on-board subsystems of the Gaia mission. Compression results for these files are shown in table 6.2. Data in Sky Mapper (SM) files corresponds to single-star images with relatively low resolution, while Astrometric Field (AF) data files have better resolution. Data from the Blue Photometers (BP), Red Photometers (RP) and from the Radial Velocity Spectrometer (RVS) resemble dispersed images with medium and high resolutions. Finally, *XP* data files contain the data of the BP and RP concatenated.

6.2.1.3 GPS

Global Positioning System (GPS) data has been included in these tests as the signal model is adequate for the proposed compression approach. The files contain raw data from a glacier GPS observation station in Greenland. The resulting data has extremely slow variations. There are three observables in these files [20]:

- C1, the pseudo-range using C/A-Code on L1.
- L1, the phase measurement on L1.
- S1, the raw signal strength or Signal to Noise Ratio (SNR) value as given by the receiver.

These observables are split into three different files, namely `global_S1`, `global_L1` and `global_C1`. A normalisation was necessary in L1 and C1 files in order to adapt the data to 24 bits. These files contain latitude, longitude and altitude with a sampling rate of 1 sample every 15 seconds. Data in `is07_*` corresponds to the glacier position while `nun2_*` consists of position coordinates from a nearby mountain. A more sophisticated data compression system for GPS data can be found in references [21] and [22].

6.2.1.4 LISA

The LISA PathFinder space mission is a technology demonstrator specifically designed to test and assess key technologies that will be used in the LISA mission. Several files

File	Size (bytes)	Compression Ratio				S_L	W_L (bits)
		Rice	Gol-Exp	Subexp	HC		
SM_L1b1t3	269000	2.38	2.46	2.49	**2.53**	2.62	16
SM_L150b0t5	1384865	1.65	1.97	1.96	**2.01**	2.14	16
SM_L90b40t5	378939	2.21	2.37	2.39	**2.42**	2.54	16
SM_L10b70t4	1248957	2.25	2.40	2.42	**2.46**	2.55	16
SM_L170b60t10	831149	2.25	2.40	2.43	**2.46**	2.57	16
AF_L1b1t3	34313	**1.78**	**1.78**	**1.78**	1.66	1.88	16
AF_L150b0t5	192171	1.32	**1.37**	1.36	1.21	1.46	16
AF_L90b40t5	56643	1.53	**1.56**	1.55	1.40	1.67	16
AF_L10b70t4	165424	1.68	**1.69**	**1.69**	1.56	1.74	16
AF_L170b60t10	118000	1.58	**1.62**	1.61	1.48	1.70	16
BP_L1b1t3	54830	2.95	2.73	**2.96**	2.85	3.00	16
BP_L150b0t5	663687	2.05	2.25	2.35	**2.40**	2.52	16
BP_L90b40t5	328485	**3.78**	3.11	3.73	**3.78**	4.00	16
BP_L10b70t4	520178	**3.19**	2.86	3.18	3.13	3.23	16
BP_L170b60t10	699466	**3.56**	3.02	3.52	3.54	3.68	16
RP_L1b1t3	55133	**2.71**	2.56	2.68	2.16	2.66	16
RP_L150b0t5	661445	2.09	2.28	2.40	**2.47**	2.59	16
RP_L90b40t5	329106	**3.77**	3.10	3.72	3.76	3.96	16
RP_L10b70t4	522874	**3.29**	2.89	3.24	3.19	3.33	16
RP_L170b60t10	696686	**3.64**	3.04	3.59	3.62	3.78	16
XP_L1b1t3	109963	**2.82**	2.64	**2.82**	2.69	2.78	16
XP_L150b0t5	1325132	2.07	2.27	2.37	**2.44**	2.54	16
XP_L90b40t5	657591	**3.77**	3.11	3.72	**3.77**	3.97	16
XP_L10b70t4	1043052	**3.24**	2.88	3.21	3.16	3.26	16
XP_L170b60t10	1396152	**3.60**	3.03	3.56	3.58	3.73	16
RVS_L1b1t3	15388	2.15	2.15	2.19	**2.28**	2.36	16
RVS_L150b0t5	165568	1.28	1.54	1.54	**1.56**	1.88	16
RVS_L90b40t5	190766	**2.36**	2.30	2.35	**2.36**	2.42	16
RVS_L10b70t4	118874	2.31	2.30	2.35	**2.42**	2.51	16
RVS_L170b60t10	281136	2.45	2.39	2.44	**2.55**	2.64	16

TABLE 6.2: Results for GIBIS simulation data files, grouped by the observation instrument.

containing all the information coming from the mission instruments are included in this corpus.

The data presented in table 6.4 are mostly related to temperature measurements. More specifically:

- Files `kp30_row5` and `kp30_row7` contain the voltage applied to the temperature control system. This control is used for the electronics tests of the mission.
- File `kp30_row10` is the output of a feed-forward filter for the temperature control of an aluminium block.

File	Size (bytes)	Compression Ratio				S_L	W_L (bits)
		Rice	Gol-Exp	Subexp	HC		
Raw GPS data							
global_S1	1716184	2.30	2.20	2.25	**2.32**	2.36	16
global_L1	3413516	1.57	1.62	**1.64**	N/A	1.77	24
global_C1	3418852	1.74	1.80	**1.82**	N/A	1.94	24
Treated data							
is07_height	28805	3.07	2.91	2.99	**3.14**	3.32	16
is07_lat	40149	**3.60**	3.00	3.56	3.58	4.41	16
is07_lon	40327	3.01	2.91	3.00	**3.15**	3.36	16
nun2_height	23014	2.99	2.86	2.92	**3.05**	3.22	16
nun2_lat	40306	3.53	3.04	3.50	**3.57**	4.18	16
nun2_lon	40306	**4.45**	4.30	**4.45**	N/A	4.93	24

TABLE 6.3: GPS data compression results, including raw files obtained from the satellite constellation and a processed data set.

File	Size (bytes)	Compression Ratio				S_L	W_L (bits)
		Rice	Gol-Exp	Subexp	HC		
kp30_row2	1154478	**3.86**	3.75	3.85	N/A	4.13	24
kp30_row3	1488555	3.90	3.85	**3.96**	N/A	4.37	24
kp30_row4	1038896	**3.44**	3.33	3.40	N/A	3.69	24
kp30_row5	1308335	**1.76**	1.73	1.74	N/A	1.81	24
kp30_row6	1206642	**1.27**	1.25	1.26	N/A	2.01	24
kp30_row7	405892	**16.70**	16.53	16.68	N/A	14.90	24
kp30_row8	1110148	**3.75**	3.62	3.72	N/A	4.03	24
kp30_row10	1292772	12.83	**15.34**	**15.34**	N/A	20.35	24
acc_intrf	8317621	**1.02**	1.00	1.00	N/A	1.22	24

TABLE 6.4: Results for LISA data files measuring temperature and position.

- Files `kp30_row8` and `kp30_row10` indicate the room temperature of the laboratory, measured with the same system to be implemented on board. Room temperature presents larger oscillations than those expected in the spacecraft.

- File `kp30_row3` represents the temperature difference between two near points. Since this difference is remarkably small, the data is essentially electronics noise.

- File `kp30_row2` contains the absolute temperature measured from a stable source.

- File `kp30_row4` refers to the absolute temperature of the measurement system. This file presents fluctuations that are not present in previous files.

- File `kp30_row6` contains the same measurements as `kp30_row3`. In this case the units are ADC counts instead of their temperature equivalent. Counts and degrees

File	Size (bytes)	Compression Ratio				S_L	W_L (bits)
		Rice	Gol-Exp	Subexp	HC		
earth_1_1500	118186	**2.83**	2.78	**2.83**	N/A	3.03	24
earth_1_5000	112479	2.15	2.19	**2.22**	N/A	2.35	24
earth_1_8500	117027	**2.09**	2.06	2.08	N/A	2.20	24
small_earthQuake	180570	**1.86**	1.80	1.83	**1.86**	1.90	16

TABLE 6.5: Seismic data files obtained from two different earthquakes.

are related exponentially, although for small variations can be considered almost linear.

- Finally, file `acc_intrf` file represents the simulation of the nominal acceleration of a test mass relative to the spacecraft. A high noise level is present in this file caused by the interferometer, which measures this acceleration.

6.2.1.5 Seismogram

Files related to seismic measurements are also present in this corpus. Table 6.5 shows the results of this analysis. All three `earth_1_*` files represent measurements of the 8.0 magnitude earthquake in Sichuan, China, occurred the May the twelfth, 2008. The three files represent the data gathered by different seismic stations. On the other hand, `small_earthQk` is a measurement of a smaller earthquake.

6.2.1.6 Spectra

The last group of data corresponds to spectroscopic measurements, which are probably the most typical instrumental data in space after imaging.

Table 6.6 summarizes the results obtained for these files. Spectra are classified into two subgroups. The first group includes large spectra with very high resolution and range for different sources. The second group involves different spectra but extremely small files, hence, low-resolution spectra. A detailed description of the files follows:

- Files `observ_irrad` and `pseudo_res_flux` contain, respectively, the observed irradiance and the pseudo-residual flux of the solar spectral atlas from 296 to 1300 nm, obtained with a Fourier Transform Spectrometer.

File	Size (bytes)	Compression Ratio				S_L	W_L (bits)
		Rice	Gol-Exp	Subexp	HC		
observ_irrad	6492192	**2.79**	2.65	2.75	2.62	2.68	16
pseudo_res_flux	7400613	**2.78**	2.70	2.75	N/A	2.74	24
spec	4463684	**1.21**	1.20	1.20	N/A	1.31	24
er_spec	3954024	**1.62**	1.60	1.61	N/A	1.63	24
all_spectra_stars	172353	2.78	2.76	**2.80**	N/A	2.81	24
all_absolute_stars	46096	**2.04**	1.97	2.01	2.00	2.08	16
all_relative_stars	35232	**2.09**	2.02	2.06	2.05	2.12	16
all_spectra_stars_16bits	170263	1.87	1.86	**1.88**	1.75	1.87	16
bkg-1o0235_freq_lin	3645	1.30	1.30	**1.31**	N/A	2.64	24
bkg-1o0235_freq_log	4585	**1.08**	1.06	1.07	N/A	2.63	24
ganimedes_freq_lin	6225	**1.31**	1.30	**1.31**	N/A	2.42	24
ganimedes_freq_log	8138	1.05	1.03	**1.06**	N/A	2.41	24
ngc1068_freq_lin	16524	**1.06**	1.03	**1.06**	N/A	2.18	24
ngc1068_freq_log	16440	**1.12**	1.11	**1.12**	N/A	2.18	24
prova_freq_lin	12453	**1.05**	1.03	1.04	N/A	2.26	24
prova_freq_log	13864	**1.06**	1.04	1.05	N/A	2.26	24

TABLE 6.6: Data compression results obtained from a variety of stellar spectra.

- Files `spec` and `er_spec` are composed of spectra and spectral error of a stellar library of 706 stars at the near-infrared spectral region (from 8348 Å to 9020 Å) with a resolution of 1.5 Å.

- File `all_spectra_stars` is the spectral energy distribution photoelectrically measured from 320 to 860 nm with a resolution of 1 nm, in equidistant steps of 1 nm. It comprises 60 bright southern and equatorial stars of intermediate and late spectral types and for all luminosity classes.

- Files `all_absolute_stars` and `all_relative_stars` represent the energy distribution for 33 galactic supergiants, mostly situated in the southern Milky Way in the range from 4800 to 7700 Å with an effective resolution of 10 Å. Specifically, `all_absolute_stars` is composed of a list of absolute fluxes for each of the 18 stars observed under good photometric conditions. The second file is a list of relative fluxes for 15 stars observed in low-quality nights.

- The second group of files (`bkg`, `ganimedes`, `ngc` and `prova`) is composed of very small low-resolution spectra in both linear and logarithmic flux scale.

	Best	Worst	Average	Std. Dev.
Exp-Gol vs CCSDS	22.47%	−16.92%	−1.57%	7.32%
Subexp vs CCSDS	21.24%	−3.31%	1.59%	4.63%
HyPER vs CCSDS	34.64%	−38.67%	1.90%	10.36%

TABLE 6.7: Relative gains in compression ratio versus the CCSDS 121.0 standard.

6.2.2 Corpus results

The first result that can be extracted from the tables presented in subsection 6.2.1 is that in almost all cases the solutions developed in this project work better than the current standard. Even in those cases where CCSDS standard delivers better compression results, differences are small.

Table 6.7 presents a statistical analysis obtained from the corpus results. Compression ratios for each coder have been normalized against the CLDCR results, thus obtaining a relative compression ratio with respect to the current standard. Values presented in this table are intended to provide a general view on each coder performance.

As seen in previous sections, both the exponential Golomb and the subexponential coders have similar performance using real data. Although the exponential Golomb coder provides robustness to the CCSDS compressor framework, it presents a performance drop when the Laplacian dispersion (b) is around 3. The subexponential coder solves this issue while slightly improving overall performance. This behavior is reflected in table 6.7, as the worst case for the exponential Golomb coder is far worse than the subexponential, namely −16.92% in front of −3.31%. Accordingly, these ratios correspond to files where the CCSDS recommendation absolute compression ratio is around 3.8 — which is the ratio obtained where the exponential Golomb coder presents a performance drop. The worst case is obtained for file `RP_L90b40t5`, where the CCSDS recommendation compression ratio is 3.77 while the exponential Golomb coder just delivers a ratio of 3.10. On the other hand, the subexponential coder delivers a ratio of 3.72, thus revealing that it solves this weakness of the exponential Golomb coder.

In general, the behavior of the subexponential coder is better than the current standard as in average it yields 1.59% better results. Additionally, the standard deviation is the smallest one, indicating highly stable results. The exponential Golomb coder efficiency is heavily penalized by the mentioned performance drop.

The HyPER coder results in table 6.7 show that the new compressor approach obtains significant gains in terms of compression. The proposed architecture provides higher average compression ratios, as well as the highest overall improvement — namely, an impressive 34.67% with respect to CCSDS 121.0. However, as previously seen in section 6.2, there are still cases where the lack of a specific stage for low entropies degrades the coder efficiency. Consequently, the standard deviation is larger than that of the previous cases, and the worst result implies a too large decrease compared to the current standard. For instance, HyPER coder delivers good ratios compared to the CCSDS 121.0 standard in the images and GIBIS data sets. Nevertheless, in files such as `for0001` and `RP_L1b1t3` it presents poor performances. In these files, the other coders use the ZB coding option as can be inferred from the compression ratios higher than the Shannon limit or close to it.

A qualitative analysis describes that the best results for the subexponential coder are delivered suitably for the GIBIS images data set. It is worth noting that this set of images is specially representative of imaging space missions. In general, the subexponential coder delivers better results for longer word lengths, as well as the HyPER coder — which presents even better results for the same data sets. The reason is that, when applied to such large sample sizes, the compressors must split the samples into two or even four parts. The most significant ones use to be mostly zero or tiny values, while the least significant ones appear as outliers (mostly flat noise). Considering the excellent behavior of our solutions in front of these cases, we can now better understand the large improvements shown in the tables.

Chapter 7

Conclusions

7.1 Conclusions

In this work two improvements to the CCSDS 121.0 standard have been developed, using exponential Golomb and subexponential coders. The aim has been to improve its compression efficiency when dealing with large amounts of noise and outliers. Additionally, a totally different approach from the current standard has been designed and implemented. This new coder is known as the HyPER coder.

As seen in chapters 3, 4 and 6 both exponential Golomb and subexponential coders provide outstanding robustness to the CCSDS compressor architecture. It has been proved that exponential coders deliver exceptional results when dealing with noisy data. However, as seen in section 3.4 the exponential Golomb coder fails to deliver a smooth start behavior in terms of code length. As a result, the low entropy options of the CCSDS standard are used beyond its intended range. This produces a severe reduction of the efficiency in this region. Therefore, the exponential Golomb is not recommended to be used within the CLDCR framework.

Subexponential codes provide a softer code length growth for low values while still maintaining short codewords for large values. This behavior solves the critical efficiency flaw, as seen in chapter 6, integrating perfectly within the CCSDS architecture. Additionally, the subexponential coder provides slightly better results than the exponential Golomb coder.

Tests conducted in this work with synthetic data reveal a general improvement when dealing with realistic levels of noise and outliers in the data. The ratios obtained are very similar on clean data, but the improvement is significant when large fractions of outliers are present. Tests with real data confirm this affirmation. In general, the proposed adaptive subexponential coder performs extremely similar to CCSDS standard, yet with typical improvements up to 20% in the compression ratio, and decreases of just 3% in the worst case.

The designed adaptive subexponential compressor offers a much more resilient operation in front of outliers and unexpected data statistics. The solution proposed here represents a very simple — yet extremely useful — improvement that can be implemented to the current standard for lossless data compression in space, offering nearly optimal compression ratios in almost any situation. It must be reminded that this coder is almost as fast as the current Rice coder as exposed in appendix A. Therefore, this thesis concludes that it is recommendable to substitute the Rice coder in CCSDS 121.0 by a subexponential coder, keeping the rest of the recommendation unchanged.

Finally, regarding the alternative method proposed using REGLIUS codes within the HyPER coder compressor structure has revealed to be promising. As seen in section 5.5, the HyPER coder maintains up to 90% efficiency levels even in very noisy environments. The segment coding strategy, similar to that of FAPEC, has proved extremely efficient when dealing with outliers. Results obtained with real data files presented in chapter 6 confirm that this new coder delivers high compression ratios in the relevant scenarios.

7.2 Future work

During the execution of this work multiple interesting research lines have been uncovered. Unfortunately, these have been out of the scope of the current work but are presented here as they might be part of future essays.

The most direct one consists in several improvement to our most recent entropy coding design. The HyPER coder is still in early development stages. Although it already yields outstanding results, some improvements might be included. For instance, for very low data entropies the HyPER coder offers a slightly worse performance than the CCSDS architecture. This is due to the lack of a specific stage for low entropies. This should

be addressed in future versions of this coder. Additionally, the current implementation only supports 16-bits codeword. Thus, future versions should be able to use different sizes — at least from 8 to 24 bits. A specific stage for low entropies is also necessary in order to improve efficiency in this range. Finally, creating an adaptive stage for the HyPER coder is necessary for this solution to be a real alternative for space missions — or even other environments. This might be similar to some extent to the one present in FAPEC. This thesis concludes encouraging additional research in this approach.

Further tests should be carried out on the performance of the coders. This is especially true for the tests carried out using synthetic data. Although the designed noise analysis can be considered a worst case scenario, tests with more realistic noise distributions might be carried out. For instance, pink noise might represent a better approximation to noise produced by some instruments.

Finally, developing a hardware implementation of the subexponential algorithm might proof useful to evaluate real performance and feasibility to be included in space missions.

Appendix A

Coders Performance

Although obtaining the best coding speed was not the primary goal of this project it was crucial to maintain an acceptable coding speed compared to the current standard. The proposed coders speed had to be similar to the current standard in order to be a feasible alternative. Therefore, in order to improve the speed of the coders the key operation for both exponential Golomb and subexponential algorithms was identified. An exhaustive profiling analysis determined that the critical operation is $\lfloor \log_2 n \rfloor$. This operation is also used to compute the length of the code from which the CCSDS compressor structure obtains the best k value. This fact increases the relevance of this operation as it is computed constantly by the compressor in order to decide the best coding option.

The $\lfloor \log_2 n \rfloor$ operation corresponds to the number of bits needed to code a given number n. A basic algorithm approach, seen in figure A.1, might be to loop and count bits until the required number of bits is found.

Although correct, this approach can be optimized. A more efficient approach is to use the number of leading zeros and the number of bits of the variable — typically known at compilation time, thus a constant in execution time [23]. Additionally, this method has

```
unsigned char rb = 1;
while (n>1) {
        n>>=1;
        ++rb;
}
return(rb);
```

FIGURE A.1: Straightforward implementation of $\lfloor \log_2 n \rfloor$.

```
return (8*sizeof(n)-__builtin_clz(n)-1);
```

FIGURE A.2: Optimized implementation of $\lfloor \log_2 n \rfloor$.

Algorithm	ns/call
Old algorithm	96.14
Using math.h	34.22
New algorithm	4.21

TABLE A.1: $\lfloor \log_2 n \rfloor$ algorithm speeds.

Coder	ns/call
Rice	16.61
Exponential-Golomb	34.13
Subexponential	22.50

TABLE A.2: Coder speeds

the advantage that most architectures have built-in assembler instructions to calculate the number of leading zeros. In order to make the code architecture independent the `gcc` function `__builtin_clz()` was used. This uses the processor assembler instruction when available or an optimized algorithm otherwise. The new proposed algorithm would be reduced to fewer operations as seen in figure A.1.

In order to evaluate the improvement obtained in this way, averaged processing times for both approaches were computed, as well as those obtained using the standard `math.h` library. The results are shown in table A.2. As expected, the new approach is extremely faster than any of the alternatives.

Although excellent results combined with an increase in the compressor robustness might be obtained with the proposed coders, maintaining coding speed within reasonable limits is essential. Exhaustive benchmarking of the coding routines has been conducted in this work to ensure that coders delivered the required performance. Table A.2 presents the average coding speeds for all the proposed coders. The exponential Golomb algorithm does not deliver the expected coding speed. This might require a hardware implementation to be in an on-board system. Otherwise, the results demonstrate that the subexponential coder speed is slightly slower than that of Rice, but still adequate and might easily fit into tight processing requirements of space-based systems.

Bibliography

[1] J. Portell, E. García–Berro, X. Luri, and A. G. Villafranca. Tailored data compression using stream partitioning and prediction: application to Gaia. *Experimental Astronomy*, 21:125–149, 2006.

[2] Consultative Committee for Space Data Systems. Lossless Data Compression, Blue Book. Technical Report CCSDS 121.0-B-1, CCSDS, 1993.

[3] P.S. Yeh. Implementation of CCSDS lossless data compression for space and data archive applications. In *Proc. Space Operations 2002 Conf.*, pages 60–69. Consultative Committee for Space Data Systems, CCSDS, 2002.

[4] R. Vitulli. PRDC: an ASIC device for lossless data compression implementing the Rice algorithm. In *Proceedings of the Geoscience and Remote Sensing Symposium, 2004.*, volume 1, pages 317–320, 2004.

[5] D. Salomon. *Data Compression. The complete reference.* Springer-Verlag, 2004.

[6] R.F. Rice. Some practical universal noiseless coding techniques. Technical Report JPL 79-22, Jet Propulsion Laboratory, Mar 1979.

[7] Consultative Committee for Space Data Systems. Lossless Data Compression, Informational Report. Technical Report CCSDS 120.0-G-2, CCSDS, 2006.

[8] P.S. Yeh, R. Rice, and W. Miller. On the optimality of code options for a universal noiseless coder. Technical Report JPL 91-2, Jet Propulsion Laboratory, Feb 1991.

[9] J. Portell, A. G. Villafranca, and E. García–Berro. Designing optimum solutions for lossless data compression in space. In *Proceedings of the On-Board Payload Data Compression Workshop 2008*, pages 35–44. ESA, 2008.

[10] J. Portell, A.G. Villafranca, and E. García-Berro. A resilient and quick data compression method of prediction errors for space missions. In Bormin Huang, Antonio J. Plaza, and Raffaele Vitulli, editors, *Proceedings of the Satellite Data Compression, Communication, and Processing V*, volume 7455, page 745505. SPIE, 2009.

[11] J. Teuhola. A compression method for clustered bit-vectors. *Inf. Process. Lett.*, 7: 308–311, 1978.

[12] D. Salomon. *Variable-length codes for data compression*. Springer, 2007.

[13] J. Wen and J.D. Villasenor. Reversible variable length codes for efficient and robust image and video coding. In *Proc. of the IEEE Data Compression Conference*, pages 65–68. IEEE, 1998.

[14] G.J. Sullivan D. Marpe, T. Wiegand. The h.264/mpeg4 advanced video coding standard and its application. *IEEE Communications Magazine*, 44:134–143, 2006.

[15] P. G. Howard and J. S. Vitter. Fast progressive lossless image compression. In M. Rabbani & R. J. Safranek, editor, *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 2186, pages 98–109, 1994.

[16] E.M. Geijo, J. Portell, E. García–Berro, X. Luri, and U. Lammers. Improved channel coding for longer contact times with Gaia. Technical Report GAIA-BCN-008, IEEC, 2004.

[17] O. Pace. Clarification on the probability of frame loss (PFL) in data transmission. Technical Report SAG-OP-001, ESA, 1998.

[18] F. Murtagh and R. H. Warmels. Test image descriptions. In *Proceedings of the first ESO/ST-ECF Data Analysis Workshop*, volume 17(6), pages 8–19. European Southern Observatory, 1989.

[19] C. Babusiaux. The Gaia Instrument and Basic Image Simulator. In C. Turon, K. S. O'Flaherty, & M. A. C. Perryman, editor, *The Three-Dimensional Universe with Gaia*, volume SP-576, pages 417–420. ESA, 2005.

[20] Y. Hatanaka. Compact RINEX format and tools. In R. E. Neilan, P. Van Scoy, and J. F. Zumberge, editor, *Proceedings of the IGS 1996*, pages 243–256. IGS, 1996.

[21] P. Ruiz. GPS data compression through an intensive study of data correlation. Master's thesis, Universitat Politècnica de Catalunya, Sep 2009.

[22] I. Mora. GPS data compression using lossless compression algorithms. Master's thesis, Universitat Politècnica de Catalunya, Sep 2009.

[23] H. S. Warren. *Hacker's delight.* Addison-Wesley, 2002.